NAVAL POSTGRADUATE SCHOOL Monterey, California



THESIS

POPULATING THE SOFTWARE DATABASE

by

Tuan Anh Nguyen

March 1996

Thesis Co-Advisors:

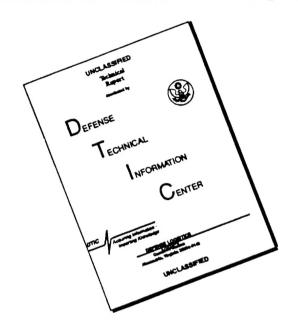
Luqi V. Berzins

Approved for public release; distribution is unlimited

DTIC QUALITY INSPECTED 1

19960517 124

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188		
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302 and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE	AND DATES COVERED	
	March 1996		Master's Thesis	
4. TITLE AND SUBTITLE	<u> </u>	5. FUNDIN	G NUMBERS	
POPULATING THE SOFTWARE D	ATABASE			
6. AUTHOR				
Nguyen, Tuan A.				
7. PERFORMING ORGANIZATION NA	ME AND ADDRESS		RMING ORGANIZATION T NUMBER	
Naval Postgraduate School Monterey, CA 93943-5000				
9. SPONSOR/MONITORING AGENCY	NAME(S) AND ADDRESS	(ES) 10. SPONS	ORING/MONITORING Y REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense of the U.S. Government.				
12a. DISTRIBUTION/AVAILABILITY STA	EMENT	12b. DIST	RIBUTION CODE	
Approved for public release; distribution is unlimited				
13. ABSTRACT (Maximum 200 words)				
existence have no standard method for selectir library components for an improved software li CAPS environment.	g components germane to the orary model proposed in [Ref f identifying candidate ret	ne intended application. [. 1]. This library would part of the components from	nts could be retrieved efficiently. The few libraries currently in This thesis focuses on the actual formation and population of provides the codes for users to implement the desired system in the Booch Ada Library - by manually inspecting over 500	
components; converting the components into a CAPS-compatible format based on the Prototyping System Description Language (PSDL) via Ada-PSDL converter program; creating algebraic specifications to match the semantic description of each component manually; and manually organizing the library into a data structure based on the multi-level filtering concept.				
This work provides (1): the base at CAPS reusable component library; (2): 75 reus	able components to be release is time intensive due to vario	ed with CAPS 95 and use ous manual processes. In	nent; (b) process of inspecting and importing components into ed to test the user interface for retrieval via multi-level filtering, specting and converting each component sometimes takes up to late this process in accordance with the base and guidelines.	
14. SUBJECT TERMS CAPS, Software Reuse, Software Base, Semantic Checking, Syntactic Checking		c Checking	15. NUMBER OF PAGES 316	
			16. PRICE CODE	
	SIFICATION OF C	ECURITY CLASSIFICATION OF	20. LIMITATION OF ABSTRACT	
REPORT THIS UNCLASSIFIED UNC	ASSIFIED	BSTRACT JNCLASSIFIED	UL	
NSN 7540-01-280-5500			Standard Form 298 (Rev 2-89) Prescribed by ANSI Std 239-18	

Approved for public release; distribution is unlimited

POPULATING THE SOFTWARE DATABASE

Tuan Anh Nguyen
Lieutenant, United States Naval Reserve
B.S.E.E., State University of New York at Buffalo, 1987

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL March 1996

Author:	Afry C
	Tuan A Nguyen
Approved by: _	Any
	Luqi, Thesis Advisor
	Naldes Bligues
_	V. Berzins, Co-Advisor
	Semi
-	T. Lewis Chairman Department of Computer Science

iv

ABSTRACT

The cost of software development could be reduced if relevant reusable software components could be retrieved efficiently. The few libraries currently in existence have no standard method for selecting components germane to the intended application. This thesis focuses on the actual formation and population of library components for an improved software library model proposed in [Ref. 1]. This library would provides the codes for users to implement the desired system in CAPS environment.

The work reported here consists of: identifying candidate reusable components from the Booch Ada Library - by manually inspecting over 500 components; converting the components into a CAPS-compatible format based on the Prototyping System Description Language (PSDL) via Ada-PSDL converter program; creating algebraic specifications to match the semantic description of each component manually; and manually organizing the library into a data structure based on the multi-level filtering concept.

This work provides (1): the base and guidelines for the (a) criteria for a reusable component; (b) process of inspecting and importing components into CAPS reusable component library; (2): 75 reusable components to be released with CAPS 95 and used to test the user interface for retrieval via multi-level filtering. The process of populating reusable components is time intensive due to various manual processes. Inspecting and converting each component sometimes takes up to an hour for each. Current tools available can be rewritten, i.e. the PSDL-Ada converter, to fully automate this process in accordance with the base and guidelines.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. WHY REUSE?	1
B. COMPUTER AIDED PROTOTYPING SYSTEM	2
C. ORGANIZATION OF CHAPTERS	3
II. BACKGROUND AND PREVIOUS RESEARCH	5
A. CAPS DESIGN AND COMPONENTS	5
B. PSDL	5
C. OBJ3 AND ALGEBRAIC SPECIFICATION	7
D. HASSE DIAGRAM	7
E. PROFILE MATCHING	9
F. CHARACTERISTICS OF A REUSABLE COMPONENT 1	0
G. SOFTWARE LIBRARIES 1	2
1. Asset Source for Software Engineering Technology (ASSET)	2
2. Reusable Ada Package for Information System Development (RAPID) 1	2
3. Common Ada Missile Package (CAMP)	3
4. Operation Support System (OSS)	3
H. METHODS OF RETRIEVAL	.3
1. Keyword Search Method	!3
2. Artificial Intelligence Methods	4
3. Multi-Level Filtering Method	14

III. DESIGN AND CONCEPTS	17
A. BOOCH LIBRARY	17
B. POPULATING PROCESS	20
IV. CONCLUSIONS AND FUTURE RESEARCH	31
A. ACCOMPLISHMENT	31
B. LESSONS LEARNED	31
C. FUTURE RESEARCH	32
1. Graphical User Interface	32
2. CAPS and the Internet	32
LIST OF REFERENCES	35
APPENDIX - LIBRARY COMPONENTS	37
INITIAL DISTRICTION LIST	307

I. INTRODUCTION

The need for code reuse has not been addressed adequately in both the academic and business world. In the business world, most organizations appear to offer primitive incentives to encourage a culture of reuse. However, very few organizations explicitly encourage programmers to reuse code, or to write code that is reused. Reuse is preached more often than it is practiced. In the academic world, the word has been used but the teaching and the practices are also limited.

One of the reasons for this is the lack of methods for effectively finding the components needed for each application and lack of component libraries organized to support such methods. With the current trend of software development, prototyping tools seem to be the key for rapid developing applications, going from design to actual implementation with executable code. This idea of reusable code is instrumental to this prototyping concept. The Department of Defense has long endorsed a programming language that is rigid in structure, for safety of operation and most important of all the reusability of codes. Ada is the standard language of the DOD culture. The purpose of this thesis is to provide a library of reusable Ada components for the Computer Aided Prototyping System (CAPS), an ongoing research project at the Naval Postgraduate School.

A. WHY REUSE?

Each year, billions of dollars are spent on computer software. Much of this effort is spent on creating and testing new source code. In order to save money, increase productivity, and improve reliability, the Department of Defense is constructing

repositories of reusable software components that can be used across applications. A great percentage of a typical program is composed of potentially reusable code [Ref. 1] and [Ref. 2]. It is desirable to make use of existing code whenever possible. This action can significantly reduce the amount of time to develop the software. With prototyping software such as CAPS, reusable code can enhance the process of rapid application development.

This approach can be summarized as follows:

- Cost savings.
- Early payback.
- Manpower savings.
- Technology leverage and risk mitigation.
- Reliability.

B. COMPUTER AIDED PROTOTYPING SYSTEM

The Computer Aided Prototyping System is a software engineering tool for developing prototype models of hard real-time embedded systems [Ref. 3] and [Ref. 6]. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototype System Description Language (PSDL), which provides facilities for modeling timing and control constraints within a software system [Ref. 4]. It is a development environment, implemented in the form of an integrated collection of tools, linked together by a user-interface as shown in Figure 1 [Ref. 5].

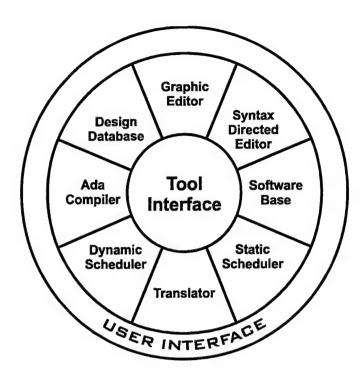


Figure 1. CAPS Functionality Overview Diagram

The library collected in this thesis is part of the Software Base component of the CAPS functionality.

C. ORGANIZATION OF CHAPTERS

Chapter II reviews the basic concepts and terms relevant to the current research of CAPS and its implementation. Chapter III focuses on the implementation of the database component of CAPS and the data structure and retrieval method for these reusable components. Chapter IV concludes the research and discusses the user interface of the software base component of CAPS.

.

.

II. BACKGROUND AND PREVIOUS RESEARCH

This chapter describes some technical background about CAPS to include PSDL.

Characteristics of reusable components and methods of retrieval are the two primary topics of this section. Various previous research and current systems are also discussed.

A. CAPS DESIGN AND COMPONENTS

CAPS is an integrated environment aimed at rapid prototyping hard real-time embedded systems [Ref. 5] and [Ref. 6]. CAPS tools include an Ada Compiler, Design Database, Graphic Editor, Syntax Directed Editor, Software Base, Static Scheduler, Dynamic Scheduler, and Translator as shown in Figure 1. Each of these components provides specific functions in the development of the software.

B. PSDL

PSDL is a text and graphics based language designed to express the specifications of real-time systems. It is based on a graphic model of vertices and edges, in which the vertices represent operators, or software processes, and the edges represent the conceptual flow of data from one operator to another. Each vertex and edge may have associated timing constraint, and the vertices may have associated control constraints.

Formally, the model used is that of an augmented graph, G = (V,E,T(v),C(v)) where G is the graph, V is the set of vertices, E is the set of edges, T(v) represents the timing constraints for the vertices, and C(v) represents the control constraints for the vertices.

Conceptually, PSDL operators may contain other operators to support the principle of abstraction. Effectively, the prototype may be expressed as a flat graph, or a

one level graph containing all the atomic operators and their streams. An atomic operator is one that is implemented in a programming language, vice a composite operator consisting of other operators and streams.

For example, the following diagram shows a PSDL prototype:

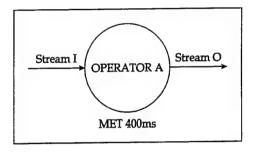


Figure 2. Example of PSDL Graph

Figure 2 represents an operation modeled by the Operator A that accepts one item from Stream I, it performs some operation on the data, and outputs Stream O. The Maximum Execution Time (MET), this is the maximum possible time the operator may take to execute the task, defined as 400 milliseconds.

Operator A can further be decomposed as shown in Figure 3 below:

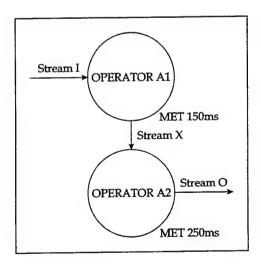


Figure 3. Decomposition of Operator A

Operator A is a composite operator, while Operator A1 and Operator A2 are atomic operators, implemented in Ada or some other language. The timing and control constraints on these atomic operators must be consistent with those of their parent operator. In a single processor the combined METs of these atomic operators cannot be greater than their parent. Operator A is really not needed for implementation of this prototype; it serves as an abstraction of the functionality of the children operators. More information about PSDL can be founded in [Ref. 8] and [Ref. 9].

C. OBJ3 AND ALGEBRAIC SPECIFICATION

OBJ3 is implemented in Common Lisp, and is based on ideas from order sorted equational logic and parameterised programming. OBJ3 provides mixfix syntax (prefix, suffix, and infix), flexible subsorts (subtypes in Ada language), parameterised modules, views, and most important term rewriting modulo associativity, commutativity, and identity. OBJ was originally designed in 1976 by Dr. Goguen [Ref. 10].

In OBJ3, an algebraic specification of objects consists of two parts: a signature and a set of axioms. The signature defines the sorts (or types) being specified, the operation symbols, and the axioms define their functionality in an object. The signature is denoted as (S, Σ) where S and Σ are a sort set and an operation symbol set, respectively. The axioms are expressed as equations describing the semantics of an object.

D. HASSE DIAGRAM

A Hasse diagram is a graphical representation of a partial ordering relation, for which the following properties hold:

- ° reflexive
- o anti-symmetric
- ° transitive

For example: the Hasse diagram for $(\{1,2,3,4\},\leq)$ is shown in Figure 4 below.

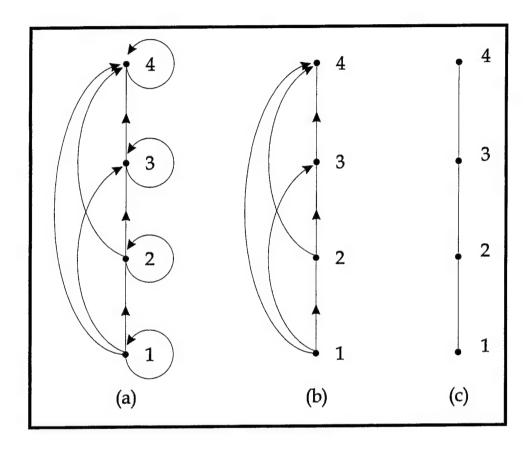


Figure 4. Constructing the Hasse Diagram for $(\{1,2,3,4\},\leq)$

This relation is called partial ordering. In Figure 4(a), the arrows indicated the relation among the members, since all members hold the reflexive property, the circle loops can be eliminated as shown in Figure 4(b), furthermore, since it is a partial ordering, all arrows implied by transitivity can be removed, as shown in Figure 4(c). This

concept can be extended to partition the software base, in which profile codes define partitions that are represented as a Hasse diagram.

E. PROFILE MATCHING

The computation for parameter matching would be very expensive if it was necessary to try all possible combinations of functions and data types with those components. For example, if a query has a function f: AAB -> B and a component has a function g: BA -> A, these two functions cannot be possibly be matched, thus there is no need to compute this combination. The purpose of profile matching is to speed up parameter matching. Profile matching is actually an efficient approximation of signature matching. A profile is a sequence of numbers that describes how data types are associated with an operation. It is defined as follows [Ref. 1]:

- ° The first integer is the total number of occurrences of sorts (data types).
- If the total number of sort groups, N > 0, then the second to $(1 + N)^{th}$ integers are the cardinalities of the sort groups, in descending order.
- The $(2 + N)^{th}$ integer is the cardinality of the unrelated sort group.
- The (3+ N)th integer is:
 0 if the value sort is different from any of the argument sorts; and
 1 if the value sort belongs to some sort group.

sort groups are bags consisting of two or more sort occurrences from the rank of the operation that are related under the relation ≡, which is the transitive-symmetric closure of the ordering ≤ on sorts.

Unrelated is a set of all sort occurrences that are not in any sort group.

sort group

For example:

Operation	Profile Code
-> A	110
AB -> C	330
AA -> B	3210
ABBCA -> C	622201
CCAAB->B	622201

Table 1. Example of Profile Code

F. CHARACTERISTICS OF A REUSABLE COMPONENT

A reusable software component should exhibit the best characteristics of any good piece of software. Specifically, it should be:

- maintainable
- efficient
- reliable
- understandable

and of course, correct. However, there are some important characteristics specific to reuse. They should have the following major characteristics:

- generality
- definiteness
- transferability
- retrievability
- sufficiency
- completeness
- primitiveness

Generality and Definiteness: for example, a component supplying elementary real functions such as max, min, floor, and ceiling is a good candidate for reusability, because these operators are well understood and are applicable to a wide range of problems; this address the issue of definiteness. However, to facilitate its reuse, we must take care to construct such a component independent of the peculiarities of any application, for example, the representation of floating-point numbers. Ideally, we should factor out such dependencies and achieve generality. The Ada language has a mechanism to implement this characteristic, namely, generics and instantiation.

Transferability and Retrievability: primarily dealing at the level of source code, not object code. Writing a component as an Ada generic package facilitates transferability, for here we have a mechanism that can capture many of the relevant parts of an abstraction. However, the management of a library with a large number of components can be a great concern. The larger the number of components the higher the cost of finding a matching component.

Sufficiency: the component captures enough characteristics of the abstraction to permit meaningful interaction with the object.

Completeness: the component interface captures all characteristics of the component. Whereas sufficiency implies a minimal collection of meaningful operations, a complete set of operations is one that covers all aspects of the underlying abstraction. For example, the abstraction of a set includes the notion of cardinality. It is not necessary to include an operation that returns the cardinality of a set; we can interact with a set without this capability. However, we should include this operation to enhance the completeness of the abstraction. Completeness is a subjective measure and in fact can be

overdone. Supplying all meaningful operations for a particular abstraction is not only overwhelming for the user, but generally unnecessary, since many high-level operations can be composed from low-level ones. For this reason, It is suggested that component operations be primitive.

Primitiveness: operations that can be implemented only with access to the underlying representation of the object. Thus, adding an item to a set is primitive, because there is no other way to implement this operation unless the underlying representation is visible. However, adding four items to a set is not primitive since it can be implemented with the adding one item iteratively [Ref. 11].

G. SOFTWARE LIBRARIES

1. Asset Source for Software Engineering Technology (ASSET)

ASSET is a software reuse library and reuse information exchange available to software developers in government, industry, and education. ASSET is sponsored by ARPA's STARS (Software Technology for Adaptable, Reliable Systems) Program to serve as a national resource for the advancement of software reuse across the DoD. The ASSET library, located in Morgantown, WV, is connected to the Internet allowing world-wide access to reusable software assets. ASSET'S goals are to create a focal point for software reuse information exchange, to advance the technology of software reuse processes and to provide an electronic marketplace for reusable software products, and stimulate a national software reuse industry.

2. Reusable Ada Package for Information System Development (RAPID)

The RAPID project is an ongoing effort in the DoD. The objective of RAPID is to provide software engineers with quick access to reusable Ada packages in the

information system domain. The system performs reusable component classification, storage and retrieval.

3. Common Ada Missile Package (CAMP)

The CAMP project is also sponsored by the DoD to create a software engineering system of reusable software library of components. The system is directed toward software for missile systems and uses Ada language for its reusable components.

4. Operation Support System (OSS)

The OSS is an ongoing project aimed at developing and integrated software engineering environment. The system is being developed at the Naval Ocean System Center. One of the goals of the project is to establish a Naval software library of reusable software components.

H. METHODS OF RETRIEVAL

1. Keyword Search Method

This is the most crude method, however simplest of all. There is no data structure in storing these components. The user, in essence, is using a primitive grep UNIX command to search for a word that associated with a component. The useful components found by this method is extremely poor when the number of components in a library is large since the set of retrieved components is relatively large. This requires the user to browse through all the found components and decide which of the components is appropriate for usage. There is no way of placing the syntactic and semantic information in this method. However, from informal survey of current programmers in the private industry, this method is very popular. This may not be a surprise due to the fact that there is no standard in retrieving reusable components.

2. Artificial Intelligence Methods

Artificial Intelligence methods include [Ref. 9] and [Ref. 10], and some recent work by Henninger [Ref. 14], which uses a knowledge base and statistical information to retrieve reusable components, based on keyword search from texts describing the components. However, because the characterization of the component behavior is completely informal, the behavior is unpredictable [Ref. 15].

3. Multi-Level Filtering Method

This method is proposed in [Ref. 1], in which a combination of retrieval processes are used. The process is represented as follows:

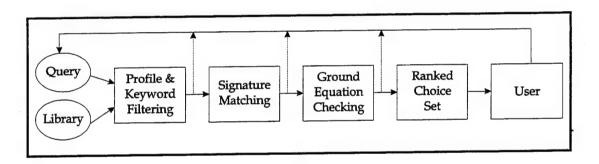


Figure 5. Model for the Multi-Level Filtering Process

In this method, search is organized as a series of increasingly stringent filters on candidate components. We first filter components by comparing their signatures with that of the query. This is accomplished by signature matching, which looks for maps that translate the type and function symbols of the query into corresponding type and function symbols of candidate components. A first stage of signature filtering can compare precomputed syntactic profiles of components with the profile of the query. These profiles are special data structures that support an efficient approximation of signature matching. The key property of a profile is that two operation signatures cannot have a syntactic

match unless their computed profiles are equal. Signature matches can be partial, in that only part of the functionality the user seeks may actually be available. The profile of an abstract data type is a bag containing the profile codes of its operations. In a partial signature match, a subset of the query profile is contained in the stored component's profile. Traditional search methods, such as keyword search, could also be used as early filters. Profile matching should be followed by full signature matching.

Semantic filters rank components by how well they satisfy the equations in the query. In this process, equations that are logical consequences of the query specification are translated through the signature matches into equations whose proof is attempted in the candidate specifications. This whole process can be made iterative.

III. DESIGN AND CONCEPTS

A. BOOCH LIBRARY

The Booch library divided into three categories: data structure, tools, and subsystems. A data structure is a component that denotes an object or class of objects characterized as an abstract state machine or an abstract data type. A tool is a component that denotes an algorithmic abstraction targeted to an object or class of objects. A subsystem is a component that denotes a logical collection of cooperating structures and tools. Each category is further divided into subcategories as shown in Figure 6 below.

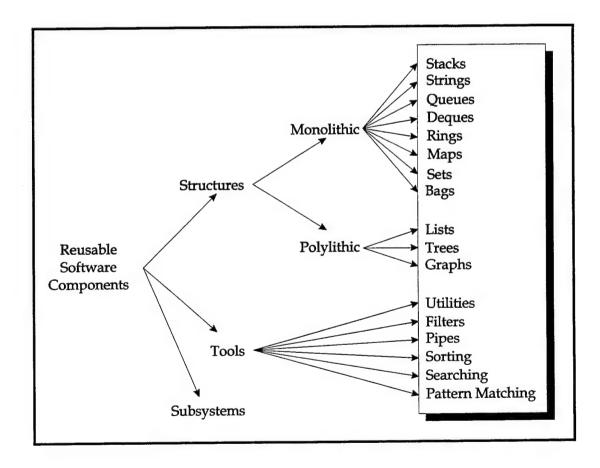


Figure 6. Booch Library

Monolithic the structure is always treated as a single unit and that individual parts of the structure can not be manipulated.

Polylithic the structure is composed of individual parts that can be manipulated.

There are over 500 components in the Booch library in many different forms. It often happens that there is a software part that we want to reuse, but it is not exactly in the right form [Ref. 16]. Figure 7 below presents the forms of reusable software component that have been found to be common across many applications [Ref. 11].

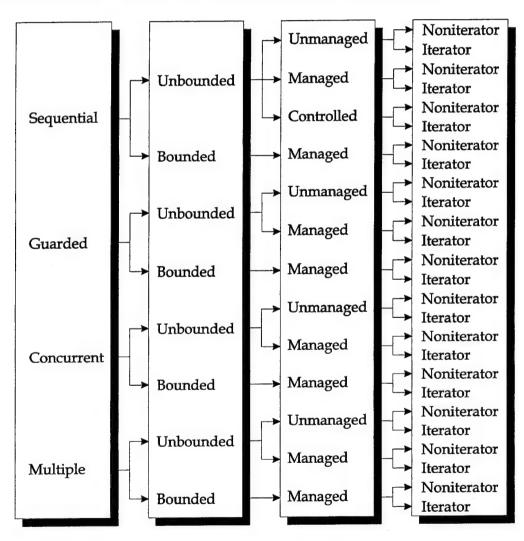


Figure 7. The forms of a reusable software component

Sequential The semantics of an object are preserved only in the presence of one thread of control for each instance of the type.

Guarded The semantics of an object are preserved in the presence of multiple threads of control, if mutual exclusion is enforced by all clients of the object.

Concurrent The semantics of an object are preserved in the presence of multiple threads of control, and mutual exclusion is enforced by the object itself.

Access by multiple clients is sequentialized.

Multiple

The semantics of an object are preserved in the presence of multiple threads of control, and mutual exclusion is enforced by the object itself.

Multiple simultaneous readers are permitted, but writers are sequentialized.

Bounded Denotes that the size of the object is static.

Unbounded Denotes that the size of the object is dynamic.

Unmanaged Automatic garbage collection is the responsibility of the underlying run time system and compiler.

Managed Garbage collection is provided by the component itself, and the type is used only by a single task.

Controlled Garbage collection is provided by the sequential component itself even if the type is used by multiple tasks.¹

Noniterator An iterator is not provided for this object.

¹ Sequential controlled means several tasks can each have a private instance of the type.

Iterator An iterator is provided for this object.

Together, these forms offer a total of 26 meaningful combinations. The Appendix lists the imported components.

The components in the library conform to the following file name convention: assuming the file name of the component is stackssbmn.

Description	File Name	
Ada specifications	vstackssbmn.a	
Ada implementation	b stackssbmn. a	
PSDL	vstackssbmn.psdl	
OBJ3 specifications	vstackssbmn.obj	
Profile code	vstackssbmn.code	

Table 2. Example of file name convention

There are 75 components imported into this library. These components are the samples of each of the data structure components in the Booch library. This should give a broad base number of the components for the reusable components.

B. POPULATING PROCESS

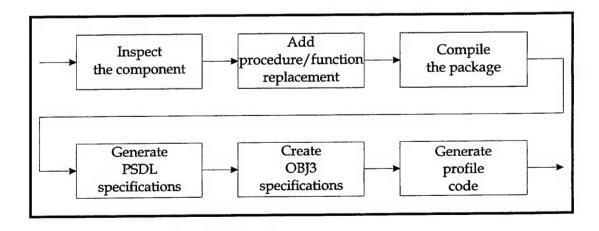


Figure 8. Populating Process

Components must be manually inspected for reusability criteria listed in Chapter II. In the CAPS system, a PSDL specification is an integrated part of a reusable component. By adding procedure versions of functions, PSDL specifications can be readily generated by a converter written by [Ref. 17].

Each step of the populating process, shown in Figure 8, is illustrated in this section by an example. An example of the first step, adding procedure/function replacement, follows:

SPECIFICATIONS

```
generic
    type Item is private;
package Stack_Sequential_Bounded_Managed_Iterator is
    type Stack(The_Size : Positive) is limited private;
   procedure Copy
                    (From_The_Stack : in
                                    : in out Stack);
                     To The_Stack
    procedure Clear (The_Stack
                                    : in out Stack);
                                    : in
                    (The_Item
                                             Item;
    procedure Push
                     On_The_Stack : in out Stack);
                                    : in out Stack);
                    (The Stack
    procedure Pop
   modified by Tuan Nguyen
   replacing functions with procedures
                                  : in Stack;
    procedure Is_Equal (Left
                        Right
                                  : in Stack;
                                  : out Boolean);
                        Result
    procedure Depth_Of (The_Stack : in Stack;
                        Result
                                   : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                                  : out Boolean);
                        Result
    procedure Top_Of (The_Stack : in Stack;
                                 : out Item);
                      Result
    end of modification
                                  : in Stack;
    function Is_Equal
                       (Left
                                  : in Stack) return
                        Right
Boolean;
    function Depth_Of (The_Stack : in Stack) return
Natural;
```

```
function Is_Empty (The_Stack : in Stack) return Boolean;
                      (The_Stack : in Stack) return Item;
    function Top_Of
    generic
        with procedure Process (The_Item : in Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Stack : in Stack);
               : exception;
    Overflow
    Underflow : exception;
private
    type Items is array(Positive range <>) of Item;
    type Stack(The_Size : Positive) is
        record
            The_Top
                      : Natural := 0;
            The Items: Items(1 .. The_Size);
        end record:
     end Stack_Sequential_Bounded_Managed_Iterator;
     IMPLEMENTATION
package body Stack_Sequential_Bounded_Managed_Iterator is
    procedure Copy (From_The_Stack : in
                                            Stack:
                    To_The_Stack : in out Stack) is
    begin
        if From The Stack.The_Top > To_The_Stack.The_Size
then
            raise Overflow;
        else
            To_The_Stack.The_Items(1 ...
From_The_Stack.The_Top) :=
                From_The_Stack.The_Items(1 ...
From_The_Stack.The_Top);
            To_The_Stack.The_Top := From_The_Stack.The_Top;
        end if:
    end Copy;
    procedure Clear (The_Stack : in out Stack) is
    begin
        The_Stack.The_Top := 0;
    end Clear;
    procedure Push (The_Item
                               : in
                    On The Stack : in out Stack) is
    begin
        On_The_Stack.The_Items(On_The_Stack.The_Top + 1) :=
The_Item;
        On_The_Stack.The_Top := On_The_Stack.The_Top + 1;
    exception
        when Constraint_Error =>
```

```
raise Overflow;
   end Push;
   procedure Pop (The_Stack : in out Stack) is
   begin
       The_Stack.The_Top := The_Stack.The_Top - 1;
   exception
       when Constraint Error =>
           raise Underflow:
   end Pop;
   modified by Tuan Nguyen
   replacing procedures with functions
   procedure Is_Equal (Left : in Stack;
                        Right : in Stack;
                        Result : out Boolean) is
   begin
       Result := Is_Equal(Left, Right);
   end Is Equal;
   procedure Depth_Of (The_Stack : in Stack;
                        Result
                                  : out Natural) is
   begin
       Result := Depth_Of(The_Stack);
   end Depth_Of;
   procedure Is_Empty (The_Stack : in Stack;
                                  : out Boolean) is
                        Result
   begin
        Result := Is_Empty(The_Stack);
   end Is Empty;
   procedure Top_Of (The_Stack: in Stack;
                                : out Item) is
                      Result
   begin
        Result := Top_Of(The_Stack);
    end Top_Of;
   end of modification
    function Is_Equal (Left : in Stack;
                       Right: in Stack) return Boolean is
    begin
        if Left.The_Top /= Right.The_Top then
            return False;
        else
            for Index in 1 .. Left.The_Top loop
                if Left.The_Items(Index) /=
Right.The_Items(Index) then
                    return False;
                end if;
            end loop;
            return True;
```

```
end if:
    end Is_Equal;
    function Depth_Of (The_Stack : in Stack) return Natural
is
    begin
        return The_Stack.The_Top;
    end Depth_Of;
    function Is_Empty (The_Stack : in Stack) return Boolean
is
        return (The_Stack.The_Top = 0);
    end Is_Empty;
    function Top_Of (The_Stack : in Stack) return Item is
        return The_Stack.The_Items(The_Stack.The_Top);
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;
    procedure Iterate (Over_The_Stack : in Stack) is
        Continue : Boolean;
    begin
        for The_Iterator in reverse 1 ...
Over The Stack. The Top loop
            Process(Over_The_Stack.The_Items(The_Iterator),
Continue):
            exit when not Continue;
        end loop;
    end Iterate;
end Stack_Sequential_Bounded_Managed_Iterator;
```

This procedure is necessary to match the code interface conventions of the current implementation of CAPS. The next step is to generate the PSDL specification for the component. The converter program will generate the PSDL automatically with the following command:

ada2psdl filename (without any extension)

The output file will have the same name as the file name with the psdl extension.

The generated file for the above example follows:

PSDL

```
TYPE Stack_Sequential_Bounded_Managed_Iterator
SPECIFICATION
 GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To The Stack: Stack
    EXCEPTIONS
      Overflow
  END
  OPERATOR Clear
  SPECIFICATION
    INPUT
      The Stack: Stack
    OUTPUT
      The_Stack : Stack
  END
  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow
  END
  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Underflow
  END
  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left: Stack,
      Right: Stack
    OUTPUT
      Result : Boolean
```

```
END
  OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
  END
  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack: Stack
    OUTPUT
      Result : Boolean
  END
  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Item
    EXCEPTIONS
      Underflow
  END
  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue
: out[t : Boolean]]
    INPUT
      Over_The_Stack : Stack
  END
END
IMPLEMENTATION ADA Stack_Sequential_Bounded_Managed_Iterator
END
```

Each procedure in Ada specifications is associated with an operator in PSDL. The input and output streams in PSDL correspond to the procedure input/output parameters.

The package must then be re-compiled for quality assurance.

OBJ3 specifications are created next in accordance with the guideline in Chapter

II. The following is an example of this step (for the previous Ada specifications):

STACK OJB3 SPECIFICATION:

```
obj STACK[X :: TRIV] is sort Stack .
  protecting NAT .
*** constructors
                               -> Stack .
   op create
                 : Stack Stack -> Stack .
   op copy
                         Stack -> Stack .
   op clear
                :
                 : Elt
                         Stack -> Stack .
   op push
                         Stack -> Stack .
   op pop
*** accessors
              : Stack Stack -> Bool .
   op isequal
                        Stack -> Nat .
   op depthof
              :
                        Stack -> Bool .
   op isempty
              :
                        Stack -> Elt .
   op topof
               :
*** exceptions
   op underflow : -> Stack .
   op underflow : -> Elt .
*** variables declaration
   var S S1 : Stack .
   var E E1 : Elt .
*** axioms
   eg clear(S) = create .
   eq copy(S,S1) = S.
   eq pop(create) = underflow .
   eq pop(push(E,S)) = S.
   eg isegual(S,S1) = S == S1 .
   eq depthof(S) = if S == create then 0
                   else 1 + depthof(pop(S)) fi .
   eq isempty(S) = S == create .
   eg topof(create) = underflow .
   eq topof(push(E,S)) = E.
endo
```

The next step is to create the profile code:

From either the Ada specifications or PSDL:

```
(From The Stack : in
                                                  Stack;
procedure Copy
                     To The Stack : in out Stack);
has the signature AB -> B
      ° first digit is the number of sort occurrences: 3
      ^{\circ} the number of sort groups is 1 thus N = 1;
      (1 + N)^{th} digit is the cardinality of the sort group
      ° second digit (1 + 1) is : 2 since |[B,B]| = 2
      ° third digit (2 + 1) is: 1 since [A] is the only unrelated sort group
      ° fourth digit (3 + 1) is: 1 since B belongs to the sort group
thus: profile(Copy) = 3211
                                 : in out Stack);
procedure Clear (The_Stack
Clear: A -> A has profile 2201
procedure Push
                    (The_Item
                                      : in
                     On_The_Stack
                                      : in out Stack);
Push: AB -> B has profile 3211
                          (The_Stack : in out Stack);
      procedure Pop
      Pop: A -> A has profile 2201
                                    : in Stack;
procedure Is_Equal (Left
                                   : in Stack;
                        Right
                                   : out Boolean);
                        Result
Is_Equal: AB -> C has profile 330
procedure Depth_Of (The_Stack : in Stack;
                                  : out Natural);
                        Result
Depth_Of: A -> B has profile 220
procedure Is_Empty (The_Stack : in Stack;
                                  : out Boolean);
                        Result
Is_Empty: A -> B has profile 220
procedure Top_Of (The_Stack : in Stack;
                      Result
                                   : out Item);
```

Top_Of: A -> B has profile 220

Summary:

Operation	Signature	Profile Code
Сору	AB -> B	3211
Clear	A -> A	2201
Push	AB -> B	3211
Pop	A -> A	2201
Is_Equal	AB -> C	330
Is_Empty	A -> B	220
Depth_Of	A -> B	220
Top_Of	A -> B	220

Table 3. Summary of Stack Profile Code

The profile codes from these components will then be partitioned and represented by a Hasse diagram to optimize the multi-filtering retrieval method.

IV. CONCLUSIONS AND FUTURE RESEARCH

This chapter summarizes the concept and the process of populating the software base. Lessons learned and suggestions for future research are also mentioned in this section.

A. ACCOMPLISHMENT

This thesis has described the process of populating the software base and relevant method for retrieval, namely, multi-level filtering concept. The components selected comprise the base library listed in the Appendix, which can be used for future study and testing of the multi-level filtering process. This process is labor intensive and many automation issues should be investigated further. Preliminary study of the retrieval has been very promising [Ref. 18].

B. LESSONS LEARNED

The process is time intensive. Not all components can be reused. The primary difference between engineering reusable components, i.e. nuts and bolts, and software engineering is continuity in dimension. A nut will be manufactured only in certain dimensions such as 5/8" but a graphical representation of a nut in software engineering can be any size.

The writing of the OBJ3 specifications associated with each component is the most difficult task of all. OBJ3 is a functional language, however Ada components are written with procedures. Thus multiple out parameters cannot be directly implemented. The rationale for using OBJ3 is to attach the semantics of the operations to each data type. By attaching this specification to a component the system can refine the retrieving

process. The user can accurately retrieve the matched component via this specification. However, the user, most of the time, does not search for an exact component, just for an approximation of the component. The user must and should inspect and modify the component found to meet his/her requirements. Thus completely detailed OBJ3 specifications may not be that critical. For example: a bounded stack will have an overflow exception in its specification. This aspect cannot be easily handled during semantic matching. Consequently, the user must supply the size parameter during instantiation. This exception can be omitted in the OBJ3 specification because the semantic matching process cannot use the information. A more appropriate treatment of the exception is to include an informal explanation sufficient to guide the user in instantiating the size bound. The informal description part of the PSDL specification can be used for this purpose.

C. FUTURE RESEARCH

1. Graphical User Interface

A graphical user interface can make the retrieval process less error prone. The user would not need to be an expert in how the software base works. This interface will increase productivity.

2. CAPS and the Internet

Currently, CAPS can be used on a local area network Unix environment or a stand alone Unix workstation. There is a plan to implement CAPS on another microprocessor base, namely, the Intel architecture microprocessor. However, CAPS can be used across platforms via the Internet. JavaScript, based on the Java language (a derivative of the C++ language), and the Internet can make this possible. JavaScript extends the

programmatic capabilities of a typical Internet browser, i.e. Netscape, to a wide range of authors and is easy enough for anyone who can compose Hyper Text Markup Language (HTML). JavaScript can be used to glue HTML, inline plug-ins, and Java applets (applications) to each other. It provides the ability to change images, play different sounds, and more in response to specified events such as a user mouse click or screen exit and entry.

The JavaScript language resembles Java, but without Java's static typing and strong type checking. JavaScript supports most of Java's expression syntax and basic control flow constructs. In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a run-time system based on a small number of primitive types. The members of numeric, boolean, and string types can be expressed literally.

Primitive types can be composed into objects by setting properties with the assignment operator. JavaScript also supports functions, again without any declarative requirements beyond the need to distinguish a function definition from other sentences in the language. Functions can be properties of objects, executing as loosely-typed methods.

JavaScript complements Java by exposing useful properties of Java applets to script authors. JavaScript scripts embedded in HTML documents can get and set exposed properties in order to query the state or alter the performance of an applet or plug-in.

Java is an extension language designed, in particular, for fast execution and type safety. (Type safety is reflected by being unable to cast a Java int into an object reference or to get at private memory by corrupting Java bytecodes). Java's strong typing also increases compilation efficiency of Java bytecode to machine code.

Java programs consist exclusively of classes and their methods. Java's requirements for declaring classes, writing methods, and ensuring type safety make programming more complex than JavaScript authoring. Java's inheritance and strong typing also tend to require tightly coupled object hierarchies.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically-typed languages like HyperTalk and Dbase. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

In summary, JavaScript can be used to implement World Wide Web access to various aspects of CAPS. For example, a graphical user interface, written in JavaScript, can enable the user to retrieve a component from the Software Base library. JavaScript can provide dialog boxes, error messages, and help systems. These features enable the user to interact with CAPS via the Internet without having to fully implement CAPS locally. Multimedia (video and audio) can be distributed over the Internet as a marketing tool for CAPS.

LIST OF REFERENCES

- 1. Joseph Goguen, Doan Nguyen, José Meseguer, Luqi, Du Zhang, Valdis Berzins. "Software Component Search", *Journal of Systems Integration*, Vol. 6, No. 1, 1996, pp. 93-134.
- 2. Cox, B. J., "Planning the Software Industrial Revolution", *IEEE Software*, September, 1990, pp. 22-30.
- 3. Luqi, V. Berzins, "Rapidly Prototyping Real-Time Sytems", *IEEE Software*, September 1988, pp. 25-36.
- 4. Luqi, "Real-Time Constraints in a Rapid Prototyping Language", Journal of Computer Languages, Vol. 18, No. 2, Spring 1993, pp. 77-103.
- 5. Jim Brockett, "The Computer Aided Prototyping System (CAPS)", A CAPS Tutorial, 1994.
- 6. Luqi and Ketabchi, M., "A Computer Aided Prototyping System", *IEEE Software*, March 1988, pp. 66-72.
- 7. Luqi, "Computer Aided Software Prototyping", *IEEE Computer*, September 1991, pp. 111-112.
- 8. Luqi, Berzins, V., Yeh, R. T., "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, October 1988, Vol. 14, No. 10, pp. 1409-1423.
- 9. Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer*, May 1989, pp. 13-25.
- 10. Joseph Goguen. "Abstract Errors for Abstract Data Types", *Proceedings, First IFIP Working Conference on Formal Description of Programming Concepts*, MIT press, 1977, pp. 21.1-21.32.
- 11. Grady Booch, "Software Components with Ada", The Benjamin/Cummings Publishing Company, Inc., 1987.
- 12. G. Fischer, Scott Henninger, and D. Redmiles. "Cognitive Tools for Locating and Comprehending Software Components", *Proceedings*, 13th International Conference on Software Engineering, May 1991.
- 13. Horowitz, E., and J. Munson. "An Expansive View of Reusable Software", *IEEE Transactions on Software Engineering*, Vol. SE-10(5), p. 479.

- 14. E. Ostertag, J. Hendler, Rubin Prieto-Diaz, and C. Braun. Computing Similarity in a Reusable Library System. *ACM Transaction on Software Engineering and Methodology*, July 1992, pp. 205-228.
- 15. Scott Henninger and Cornelia Boldyreff. "Software Engineer's Reference Book". Butterworth-Heinemann, 1991.
- 16. Goguen, J. "Parameterized Programming", *IEEE Transactions on Software Engineering*, Vol. SE-10(5), September 1984, pp. 474.
- 17. Christopher S. Eagle, "Tools for Storage and Retrieval of Ada Software Components in a Software Base", Master Thesis, March 1995.
- 18. Doan Nguyen, "Architectural Model for Software Component Search", Ph. D. Dissertation, December 1995.

APPENDIX - LIBRARY COMPONENTS

BOOCH LIBRARY COMPONENTS

The following lists are grouped by major component class.

Bags

1	Bag_Simple_Sequential_Bounded_Managed_Iterator
2	Bag_Simple_Sequential_Bounded_Managed_Noniterator
3	Bag_Simple_Sequential_Unbounded_Managed_Iterator
4	Bag_Simple_Sequential_Unbounded_Managed_Noniterator
5	Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator
6	Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator

Lists

1	List_Double_Bounded_Managed
2	List_Double_Unbounded_Managed
3	List_Double_Unbounded_Unmanaged
4	List_Single_Bounded_Managed
5	List_Single_Unbounded_Managed
6	List_Single_Unbounded_Unmanaged

Maps

1	Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator
2	Map_Simple_Noncached_Sequential_Bounded_Managed_Noniterator
3	Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
4	Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator
5	Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator

Queues

1	Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator
2	Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator
3	Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator
4	Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
5	Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator
6	Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator
7	Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator
8	Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
9	Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
10	Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
11	Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator
12	Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
13	Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator
14	Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
15	Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
16	Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
17	Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
18	Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator

Rings

1	Ring_Sequential_Bounded_Managed_Iterator
2	Ring_Sequential_Bounded_Managed_Noniterator
3	Ring_Sequential_Unbounded_Managed_Iterator
4	Ring_Sequential_Unbounded_Managed_Noniterator
5	Ring_Sequential_Unbounded_Managed_Iterator
6	Ring_Sequential_Unbounded_Managed_Noniterator

Sets

1	Set_Simple_Sequential_Bounded_Managed_Iterator
2	Set_Simple_Sequential_Bounded_Managed_Noniterator
3	Set_Simple_Sequential_Unbounded_Managed_Iterator
4	Set_Simple_Sequential_Unbounded_Managed_Noniterator
5	Set_Simple_Sequential_Unbounded_Unmanaged_Iterator
6	Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator

Sorts & Searchs

1	Binary_Search
2	Binary_Insertion_Search
3	Buble_Sort
4	Heap_Sort
5	Natural_Merge_Sort
6	Ordered_Sequential_Search
7	Poly_Sort
8	Quick_Sort
9	Radix_Sort
10	Sequential_Search
11	Shaker_Sort
12	Shell_Sort
13	Straight_Insertion_Sort
14	Straight_Selection_Sort

Stacks

1	Stack_Sequential_Bounded_Managed_Iterator
2	Stack_Sequential_Unbounded_Managed_Noniterator
3	Stack_Sequential_Unbounded_Managed_Iterator
4	Stack_Sequential_Unbounded_Unmanaged_Noniterator
5	Stack_Sequential_Unbounded_Unmanaged_Iterator

Storage

1	Storage_Sequence	

Strings

1	String_Sequential_Unbounded_Controlled_Iterator
2	String_Sequential_Unbounded_Managed_Iterator
3	String_Sequential_Bounded_Unmanaged_Noniterator
4	String_Sequential_Unbounded_Unmanaged_Noniterator

Trees

1	Tree_Arbitrary_Double_Bounded_Unmanaged
2	Tree_Arbitrary_Double_Unbounded_Unmanaged
3	Tree_Arbitrary_Single _Bounded_Unmanaged
4	Tree_Arbitrary_Single _Unbounded_Unmanaged

BAG OBJ3 SPECIFICATIONS

```
obj BAG[X :: TRIV] is sort Bag .

protecting NAT .

*** constructors

op create : -> Bag Bag -> Bag op copy : Bag Bag -> Bag op add : Elt Bag -> Bag op add : Elt Bag -> Bag op union : Bag Bag Bag -> Bag op intersection : Bag Bag Bag -> Bag op difference : Bag Bag Bag -> Bag op difference : Bag Bag Bag -> Bag op difference : Bag Bag Bag -> Bag op intersection : Bag Bag Bag -> Bag op difference : Bag Bag Bag -> Bag op intersection : Bag Bag Bag -> Bag op intersection : Bag Bag Bag -> Bag op intersection : Bag Bag Bag -> Bag intersection : Bag -> Nat intersection : Bag -> Bool intersection : Bag -> B
```

```
eq clear(B) = create .
eq remove(E,create) = itemisnotinbag .
eq remove(E,add(E1,B1)) = if E == E1 then B1 else
add(E1,remove(E,B1)) fi .
eq union(B,create,B1) = B .
eq union(B,create,B1) = B .
eq union(B,add(E1,B1),B2) = add(E1,union(B,B1,B2)) .

eq intersection(B,create,B1) = create .
eq intersection(B,add(E1,B1),B2) = if isamember(E1,B) then
add(E1,intersection(B,B1,B2)) else intersection(B,B1,B2) fi .

eq difference(B,create,B1) = B .
eq difference(Create,B,B1) = B .
eq difference(Create,B,B1) = B .
eq difference(E,add(E1,B1),B2) = if isamember(E1,B) then
difference(remove(E1,B),B1,B2) else add(E1,difference(B,B1,B2)) fi .

eq extentof(create) = 0 .
eq extentof(add(E,B)) = 1 + extentof(B) .

eq uniqueextentof(add(E,B)) = if isamember(E,B) then
uniqueextentof(B) else 1 + uniqueextentof(B) fi .

eq isempty(B) = B == create .
eq isamember(E,create) = false .
eq isamember(E,create) = false .
eq isamember(E,create,B) = true .
eq isasubset(add(E,B),B1) = if isamember(E,B1) then isasubset(B,B1)
else false fi .

eq isapropersubset(B,B1) = isasubset(B,B1) and extentof(B1) >
extentof(B) .
endo
```

BAGS PROFILE CODES

OPERATORS	SIGNATURES	PROFILE CODES
COPY	A B -> B	3211
CLEAR	A -> A	2201
ADD	A B -> B	3211
REMOVE	A B -> B	3211
UNION	A B C -> C	4231
INTERSECTION	A B C -> C	4231
DIFFERENCE	ABC->C	4231
IS_EQUAL	A B -> C	330
EXTENT_OF	A -> B	220
UNIQUE_EXTENT_OF	A -> B	220
IS EMPTY	A -> B	220
IS A MEMBER	A B -> C	330
IS A SUBSET	A B -> C	330
IS A PROPER_SUBSET	A B -> C	330

SET OF PROFILE: {4231,3211,2201,330,220}

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
eq clear(B) = create .
eq remove(E, create) = itemisnotinbag .
eq remove(E, add(EI,B1)) = if E == E1 then B1 else
add(E1,remove(E,B1)) fi .

eq union(B, create,B1) = B .
eq union(B, add(E1,B1),B2) = add(E1,union(B,E1,B2)) .

eq intersection(B, create,B1) = create .
eq intersection(B, add(E1,B1),B2) = if isamember(E1,B) then
add(E1,intersection(B,B1,B2)) else intersection(B,B1,B2) fi .

eq difference(B, create,B,B1) = B .
eq difference(Create,B,B1) = B .
eq difference(Create,B,B1),B2) = if isamember(E1,B) then
difference(remove(E1,B),B1,B2) else add(E1,difference(B,B1,B2)) fi .

eq extentof(create) = 0 .
eq extentof(add(E,B1)) = 1 + extentof(B) .

eq uniqueextentof(create) = 0 .
eq uniqueextentof(B) else 1 + uniqueextentof(B) fi .

eq isamember(E, create) = false .
eq isamember(E, create) = false .
eq isamember(E, add(E1,B1)) = E == E1 or isamember(E,B1) .

eq isasubset(create,B) = true .
eq isasubset(create,B) = true .
eq isasubset(create,B) = if isamember(E,B1) then isasubset(B,B1)
else false fi .

eq isapropersubset(B,B1) = isasubset(B,B1) and extentof(B1) >
extentof(B) .
endo
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"

-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body Bag_Simple_Sequential_Bounded_Managed_Iterator is
       procedure Copy (From_The_Bag : in Bag;
To_The_Bag : in out Bag) is
              if From_The_Bag.The_Back > To_The_Bag.The_Size then
  raise Overflow;
                      To_The_Bag.The_Items(1 .. From_The_Bag.The_Back) := From_The_Bag.The_Items(1 .. From_The_Bag.The_Back);
To_The_Bag.The_Back := From_The_Bag.The_Back;
               end if:
        end Copy;
       procedure Clear (The_Bag : in out Bag) is begin_
       The_Bag.The_Back := 0;
end Clear;
       procedure Add (The_Item : in Item;
To_The_Bag : in out Bag) is
       begin
              for Index in 1 .. To_The_Bag.The_Back loop
   if The_Item = To_The_Bag.The_Items(Index).The_Item then
        To_The_Bag.The_Items(Index).The_Count :=
              To_The_Bag.The_Items(Index).The_Count + 1;
                             return:
              end if;
end loop;
To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Item :=
The Item:
              n;
To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Count := 1;
To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
        exception
       raise Overflow;
end Add;
              when Constraint_Error =>
       procedure Remove (The_Item : in Item; From_The_Bag : in out Bag) is
              in
for Index in 1 .. From_The_Bag.The_Back loop
   if The_Item = From_The_Bag.The_Items(Index).The_Item then
        if From_The_Bag.The_Items(Index).The_Count > 1 then
            From_The_Bag.The_Items(Index).The_Count :=
            From_The_Bag.The_Items(Index).The_Count - 1;
                             else From_The_Bag.The_Items(Index .. (From_The_Bag.The_Back -
1)) :=
                                   end if;
                      return;
end if;
               end loop;
raise Item_Is_Not_In_Bag;
       end Remove:
       procedure Union (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
              To Index : Natural;
               To_Back : Natural;
              while To_Index > 0 loop
if To_The_Bag.The_Items(To_Index).The_Item =
And_The_Bag.The_Items(And_Index).The_Item then
                                    exit:
                             To_Index := To_Index - 1;
end if;
                     end 1r;
end loop;
if To_Index = 0 then
    To_The_Bag.The_Items(To_The_Bag.The_Back + 1) :=
    And_The_Bag.The_Items(And_Index);
```

```
To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
                                      To_The_Bag.The_Items(To_Index).The_Count :=
To_The_Bag.The_Items(To_Index).The_Count +
And_The_Bag.The_Items(And_Index).The_Count;
                   end if;
end loop;
          exception
                   when Constraint Error =>
          end Union;
         procedure Intersection (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
                   And_Index : Natural;
                  Inc. The_Bag. The_Back := 0;

TO_The_Bag. The_Back := 0;

For Of_Index in 1 .. Of_The_Bag. The_Back loop

And_Index := And_The_Bag. The_Back;

while And_Index > 0 loop

if Of_The_Bag. The_Items(Of_Index). The_Item =

And_The_Bag. The_Items(And_Index). The_Item then

if Of_The_Bag. The_Items(Of_Index). The_Count <

And_The_Bag. The_Items(And_Index). The_Count then

To_The_Bag. The_Items(To_The_Bag. The_Back +

The_Items(To_The_Bag. The_Back)

The_Items(To_The_Bag. The_Back)

The_Items(To_The_Bag. The_Back)
          begin
1).The Item
                                                        := Of_The_Bag.The_Items(Of_Index).The_Item;
To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The Count
                                                        := Of_The_Bag.The_Items(Of_Index).The_Count;
To_The_Bag.The_Back := To_The_Bag.The_Back +
                                               else
   To_The_Bag.The_Items(To_The_Bag.The_Back +
1) The Item
                                                       := Of_The_Bag.The_Items(Of_Index).The_Item;
To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The Count
:=
And_The_Bag.The_Items(And_Index).The_Count;
To_The_Bag.The_Back := To_The_Bag.The_Back +
                                               end if;
                                              exit:
                                      else
         else
And_Index := And_Index - 1;
end if;
end loop;
end loop;
exception
when Constraint_Error =>
          raise Overflow;
end Intersection;
         procedure Difference (Of_The_Bag : in
                                                            And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
                  And Index : Natural:
         begin
                  in
To_The_Bag.The_Back := 0;
for Of_Index in 1 .. Of_The_Bag.The_Back loop
And_Index := And_The_Bag.The_Back;
while And_Index > 0 loop
    if Of_The_Bag.The_Items(Of_Index).The_Item =
        And_The_Bag.The_Items(And_Index).The_Item then
                            Of_The_Bag.The_Items(Of_Index).The_Item;
To_The_Bag.The_Items(To_The_Bag.The_Back +
                 Of The Bag. The Items (To The Bag. The Back +

Of The Bag. The Items (Of Index). The Count -
And The Bag. The Items (And Index). The Count;

To The Bag. The Back := To The Bag. The Back + 1;
end if;
end loop;
sption
1).The_Count :=
          exception
when Constraint_Error =>
raise Overflow;
          end Difference;
         modified by Tuan Nguyen and Vincent Hong
         date: 8 April 1995
adding procedures to replace functions
         procedure Is_Equal (Left
                                                       (Left : in Bag;
Right : in Bag;
```

```
Result : out Boolean) is
begin
Result := Is_Equal(Left, Right);
end Is_Equal;
begin
    Result := Extent_Of(The_Bag);
end Extent_Of;
Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;
procedure Number_Of (The_Item : in Item;
In_The_Bag : in Bag;
Result : out Positive) is
        Result := Number_Of(The_Item,In_The_Bag);
end Number_Of;
Result := Is_Empty(The_Bag);
end Is_Empty;
procedure Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag;
Result : out Boolean) is
Result := Is_A_Member(The_Item,Of_The_Bag);
end Is_A_Member;
procedure Is_A_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
Result := Is_A_Subset(Left,Right);
end Is_A_Subset;
procedure Is_A_Proper_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
end of modification
if Left.The_Back /= Right.The_Back then
    return False;
    else
         for Left_Index in 1 .. Left.The_Back loop
             else
                      exit;
end if;
                  else
                      Right_Index := Right_Index - 1;
                 end if;
             end loop;
if Right_Index = 0 then
    __ kight_Index = t
return False;
end if;
end loop;
return True;
end if;
Is P--
end Is_Equal;
Count : Natural := 0;
begin
    for Index in 1 .. The_Bag.The_Back loop
        Count := Count + The_Bag.The_Items(Index).The_Count;
end loop;
    return Count;
end Extent_Of;
function Unique_Extent_Of (The_Bag : in Bag) return Natural is
return The_Bag.The_Back;
end Unique_Extent_Of;
function Number_Of (The_Item : in Item;
In_The_Bag : in Bag) return Positive is
```

```
begin
            n
for Index in 1 .. In_The_Bag.The_Back loop
    if The_Item = In_The_Bag.The_Items(Index).The_Item then
        return In_The_Bag.The_Items(Index).The_Count;
    end if;
     end loop;
end loop;
raise Item_Is_Not_In_Bag;
end Number_Of;
      function Is_Empty (The_Bag : in Bag) return Boolean is
      begin
    return (The_Bag.The_Back = 0);
end Is_Empty;
     function Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag) return Boolean is
           in
  for Index in 1 .. Of_The_Bag.The_Back loop
    if Of_The_Bag.The_Items(Index).The_Item = The_Item then
        return True;
  end if;
end loop;
return False;
      end Is_A_Member;
     begin
            in
for Left_Index in 1 .. Left.The_Back loop
  Right_Index := Right.The_Back;
  while Right_Index > 0 loop
    if Left.The_Items(Left_Index).The_Item =
        Right.The_Items(Right_Index).The_Item then
                               exit;
                        else
                  Right_Index := Right_Index - 1;
end if;
end loop;
if Right_Index = 0 then
                   return False;
elsif Left.The_Items(Left_Index).The_Count >
Right.The_Items(Right_Index).The_Count then
                        return False:
                   end if
     end 11;
end loop;
return True;
end Is_A_Subset;
     begin
            in
for Left_Index in 1 .. Left.The_Back loop
Right_Index := Right.The_Back;
while Right_Index > 0 loop
if Left.The_Items(Left_Index).The_Item =
    Right.The_Items(Right_Index).The_Item then
                        exit:
else
Right_Index := Right_Index - 1;
end if;
                   end loop;
if Right_Index = 0 then
                  return False;
elsif Left.The_Items(Left_Index).The_Count >
Right.The_Items(Right_Index).The_Count then
return False;
                  end loop;
if Left.The_Back < Right.The_Back then
            return True;
elsif Left.The_Back > Right.The_Back then
return False;
else
                  return (Total_Left_Count < Total_Right_Count);
             end if;
      end Is_A_Proper_Subset;
      procedure Iterate (Over_The_Bag : in Bag) is
   Continue : Boolean;
      begin
            for The_Iterator in 1 .. Over_The_Bag.The_Back loop
Process(Over_The_Bag.The_Items(The_Iterator).The_Item,
Over_The_Bag.The_Items(The_Iterator).The_Count,
     exit when not Continue;
end loop;
end Iterate;
Continue);
```

end Bag_Simple_Sequential_Bounded_Managed_Iterator;

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Bounded_Managed_Iterator SPECIFICATION
  GENERIC
  GENERIC
Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
From_The_Bag : Bag,
To_The_Bag : Bag
      OUTPUT
To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Clear SPECIFICATION
      INPUT
         The_Bag : Bag
      OUTFUT
The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  END
  OPERATOR Add
SPECIFICATION
      INPUT
The_Item : Item,
To_The_Bag : Bag
      OUTPUT
      To_The_Bag : Bag
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
  END
  OPERATOR Remove
SPECIFICATION
      INPUT
         The_Item : Item,
      From_The_Bag : Bag
OUTPUT
From_The_Bag : Bag
      PYCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Union SPECIFICATION
      INPUT
         NPUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
      OUTPUT
To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Intersection
   SPECIFICATION
      PECIFICATION

INFUT

Of The Bag: Bag,
And The Bag: Bag,
To The Bag: Bag

OUTPUT

To The Bag: Bag

EXCEPTIONS

OUTPUT THE BAG
         Overflow, Item_Is_Not_In_Bag
   END
  OPERATOR Difference
SPECIFICATION
INPUT
Of_The_Bag : Bag,
      Or_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Is Equal
   SPECIFICATION
INPUT
Left: Bag,
```

Right : Bag

```
OUTPUT
Result : Boolean
EXCEPTIONS
                    Overflow, Item_Is_Not_In_Bag
      OPERATOR Extent_Of
       SPECIFICATION
             INPUT
                    The_Bag : Bag
             OUTPUT
Result : Natural
             EXCEPTIONS
                    Overflow, Item_Is_Not_In_Bag
      OPERATOR Unique_Extent_Of SPECIFICATION
             The_Bag : Bag
OUTPUT
             INPUT
                    Result : Natural
             EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
      END
      OPERATOR Is_Empty
SPECIFICATION
             INPUT
The_Bag : Bag
OUTPUT
                    Result : Boolean
             EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
      OPERATOR IS_A_Member
SPECIFICATION
INPUT
The_Item : Item,
Of_The_Bag : Bag
             OUTPUT
Result : Boolean
EXCEPTIONS
                     Overflow, Item_Is_Not_In_Bag
      OPERATOR Is_A_Subset SPECIFICATION
              INPUT
                    Left : Bag,
Right : Bag
             OUTPUT
              Result : Boolean
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
      OPERATOR Is_A_Proper_Subset
SPECIFICATION
              INPUT
                    Left : Bag,
Right : Bag
             OUTPUT
             Result : Boolean
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
      END
      OPERATOR Iterate SPECIFICATION
Process: PROCEDURE[The_Item : in[t : Item], The_Count : in[t : Positive], Continue : out[t : Boolean]]

INPUT

Output The Transfer of the Tran
              Over_The_Bag : Bag
EXCEPTIONS
                     Overflow, Item_Is_Not_In_Bag
       END
                                                      (Bag, Simple, Sequential, Bounded, Managed, Iterator)
DESCRIPTIONS:
 IMPLEMENTATION ADA Bag_Simple_Sequential_Bounded_Managed_Iterator
```

END

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
type Item is private;
package Bag_Simple_Sequential_Bounded_Managed_Noniterator is
       type Bag(The_Size : Positive) is limited private;
                                              (From The Bag : in
      procedure Copy
                                             (From_The_Bag
To_The_Bag
(The_Bag
(The_Item
To_The_Bag
(The_Item
From_The_Bag
(Of_The_Bag
                                                                    : in out Bag);
: in out Bag);
: in Item;
      procedure Clear
       procedure Add
                                                                        in out Bag);
      procedure Remove
                                                                       in
in out
                                                                                   Item;
Bag);
      procedure Union
                                                                        in
                                                                                   Bag:
     And_The_Bag
procedure Intersection (Of_The_Bag
                                                                                   Bag;
Bag;
                                                                        in out
                                                                        in
                                             And_The_Bag
To_The_Bag
(Of_The_Bag
                                                                        in
                                                                                   Bag
                                                                       in out
      procedure Difference
                                               And_The_Bag
To_The_Bag
                                                                       in
                                                                    : in out Bag);
      modified by Tuan Nguyen and Vincent Hong
date: 7 April 1995
adding procedures to replace functions
       procedure Is_Equal
                                                       (Left
                                                                           : in Bag;
: in Bag;
                                                       Right
Result
(The_Bag
Result
                                                                              out Boolean);
in Bag;
out Natural);
in Bag;
      procedure Extent_Of
                                                                           : in Bag;

: out Natural);

: in Bag;

: out Boolean);

: in Item;

: in Bag;

: out Boolean);

: in Bag;
       procedure Unique_Extent_Of
                                                       (The_Bag
                                                       Result
(The_Bag
Result
      procedure Is_Empty
                                                       (The_Item
Of_The_Bag
Result
(Left
       procedure Is_A_Member
      procedure Is_A_Subset
```

```
: in Bag;
: out Boolean);
: in Bag;
                                                   Right
                                                    Result
      procedure Is_A_Proper_Subset
                                                  (Left
                                                    Right
                                                                       in Bag
                                                    Result
                                                                     : out Boolean);
      end of modification
                                                                   : in Bag;
: in Bag)
                                                 (Left
      function Is Equal
                                                 Right
(The_Bag
(The_Bag
                                                                                   return Boolean;
return Natural;
                                                                     in Bag)
in Bag)
in Bag)
in Item;
      function Extent_Of
      function Unique_Extent_Of
function Number_Of
                                                 (The Item
                                                  In_The_Bag : in Bag)
                                                                                   return
Positive:
                                                                     in Bag)
in Item;
in Bag)
in Bag;
      function Is_Empty
function Is_A_Member
                                                 (The Bag
                                                                                   return Boolean;
                                                (The_Bag
(The_Item
Of_The_Bag
(Left
Right
(Left
                                                                                   return Boolean;
      function Is A Subset
                                                                                  return Boolean;
                                                                      in Bag)
      function Is_A_Proper_Subset
                                                                      in Bag
                                                  Right
                                                                                  return Boolean;
      Overflow : exception; Item_Is_Not_In_Bag : exception;
private
type Node is
            record
      record
The_Item : Item;
The_Count : Positive;
end record;
type Items is array(Positive range <>) of Node;
type Bag(The_Size : Positive) is
            record
            The_Back : Natural := 0;
The_Items : Items(1 .. The_Size);
end record;
end Bag_Simple_Sequential_Bounded_Managed_Noniterator;
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                   "Restricted Rights Legend"
-- Was, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body Bag_Simple_Sequential_Bounded_Managed_Noniterator is
     procedure Copy (From_The_Bag : in Bag;
To_The_Bag : in out Bag) is
           if From_The_Bag.The_Back > To_The_Bag.The_Size then
  raise Overflow;
           else
                 To_The_Bag.The_Items(1 .. From_The_Bag.The_Back) :=
    From_The_Bag.The_Items(1 .. From_The_Bag.The_Back);
To_The_Bag.The_Back := From_The_Bag.The_Back;
            end if:
     end Copy;
        rocedure Clear (The Bag : in out Bag) is
     begin
            The_Bag.The_Back := 0;
      end Clear;
     procedure Add (The_Item : in Item;
    To_The_Bag : in out Bag) is
      begin
           for Index in 1 .. To_The_Bag.The_Back loop
   if The_Item = To_The_Bag.The_Items(Index).The_Item then
        To_The_Bag.The_Items(Index).The_Count :=
        To_The_Bag.The_Items(Index).The_Count + 1;
                       return:
           return;
end if;
end loop;
To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Item :=
The_Item:
            1;
To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Count := 1;
To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
      exception
            when Constraint Error =>
     raise Overflow;
end Add;
     procedure Remove (The_Item : in Item; From_The_Bag : in out Bag) is
           else
                             From_The_Bag.The_Items(Index ..
(From_The_Bag.The_Back -
1)) :=
                             end if:
            end if;
end loop;
            raise Item Is Not In Bag;
      end Remove;
     procedure Union (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
            To_Index : Natural;
            To_Back : Natural;
           exit:
                       else
                        To_Index := To_Index - 1;
end if;
                  end loop;
if To_Index = 0 then
                        O_INCEX = V CHEN
TO_The_Bag.The_Items(To_The_Bag.The_Back + 1) :=
And_The_Bag.The_Items(And_Index);
```

```
To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
                                   To_The_Bag.The_Items(To_Index).The_Count :=
To_The_Bag.The_Items(To_Index).The_Count +
And_The_Bag.The_Items(And_Index).The_Count;
                 end if;
end loop;
         exception
when Constraint_Error =>
raise Overflow;
         end Union:
        procedure Intersection (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
                 And Index : Natural;
        begin
To_The_Bag.The_Back := 0;
for Of_Index in 1 . . Of_The_Bag.The_Back loop
And_Index := And_The_Bag.The_Back;
And_Index := And_The_Bag.The_Back;
And_Index := Oloop

'Of_Index'.The,
                          And_Index := And_The_Bag.The_Back;
while And_Index > 0 loop
   if Of_The_Bag.The_Items(Of_Index).The_Item =
        And_The_Bag.The_Items(And_Index).The_Item then
        if Of_The_Bag.The_Items(Of_Index).The_Count <
            And_The_Bag.The_Items(And_Index).The_Count then
            To_The_Bag.The_Items(To_The_Bag.The_Back +</pre>
1). The Item
                                                     := Of_The_Bag.The_Items(Of_Index).The_Item;
To_The_Bag.The_Items(To_The_Bag.The_Back +
1) .The_Count
                                                     := Of_The_Bag.The_Items(Of_Index).The_Count;
To_The_Bag.The_Back := To_The_Bag.The_Back +
                                                      To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Item
                                                     := Of_The_Bag.The_Items(Of_Index).The_Item;
To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Count
and_The_Bag.The_Items(And_Index).The_Count;
To_The_Bag.The_Back := To_The_Bag.The_Back +
                else And_Index := And_Index - 1;
end if;
end loop;
end loop;
eption
        exception when Constraint_Error =>
        raise Overflow;
end Intersection;
        procedure Difference (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
                  And_Index : Natural;
        begin
                 in
To_The_Bag.The_Back := 0;
for Of_Index in 1 .. Of_The_Bag.The_Back loop
And_Index := And_The_Bag.The_Back;
while And_Index > 0 loop
    if Of_The_Bag.The_Items(Of_Index).The_Item =
        And_The_Bag.The_Items(And_Index).The_Item then
                                            exit;
                                   And_Index := And_Index - 1;
end if;
                          end if;
end loop;
if And_Index = 0 then
   To_The_Bag.The_Items(To_The_Bag.The_Back + 1) :=
        Of_The_Bag.The_Items(Of_Index);
   To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
elsif Of_The_Bag.The_Items(Of_Index).The_Count >
        And_The_Bag.The_Items(And_Index).The_Count then
   To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Item
                                   Of_The_Bag.The_Items(Of_Index).The_Item;
To_The_Bag.The_Items(To_The_Bag.The_Back +
                Of The Bag. The Ltems (To The Bag. The Back +

Of The Bag. The Ltems (Of Index). The Count -
And The Bag. The Ltems (And Index). The Count;

To The Bag. The Back := To The Bag. The Back + 1;
end if;
end loop;
eption
1) The Count :=
        exception
when Constraint_Error =>
raise Overflow;
end Difference;
        modified by Tuan Nguyen and Vincent Hong date: 8 April 1995
        adding procedures to replace functions
```

```
Result : out Boolean) is
begin
     Result := Is_Equal(Left,Right);
end Is_Equal;
Result := Extent_Of(The_Bag);
end Extent_Of;
Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;
procedure Number_Of (The_Item : in Item;
	In_The_Bag : in Bag;
	Result : out Positive) is
          Result := Number_Of(The_Item,In_The_Bag);
end Number_Of;
procedure Is_Empty (The_Bag : in Bag;
Result : out Boolean) is
begin
Result := Is_Empty(The_Bag);
end Is_Empty;
procedure Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag;
Result : out Boolean) is
begin
     Result := Is_A_Member(The_Item,Of_The_Bag);
end Is_A_Member;
procedure Is_A_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
begin
Result := Is_A_Subset(Left,Right);
end Is_A_Subset;
procedure Is_A_Proper_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
end of modification
begin
if Left.The Back /= Right.The_Back then
     return False;
else
          for Left_Index in 1 . Left.The_Back loop
   Right_Index := Right.The_Back;
   while Right_Index > 0 loop
   if Left.The_Items(Left_Index).The_Item =
        Right.The_Items (Right_Index).The_Item then
        if Left.The_Items (Left_Index).The_Count /=
        Right.The_Items (Right_Index).The_Count then
        return False;
   else
   exit:
                          exit;
end if;
                    Right_Index := Right_Index - 1;
end if;
                end 1;
end loop;
if Right_Index = 0 then
return False;
end if;
     end if;
end loop;
return True;
end if;
end Is_Equal;
function Extent_Of (The_Bag : in Bag) return Natural is
Count : Natural := 0;
begin
     return Count;
end Extent_Of;
function Unique_Extent_Of (The_Bag : in Bag) return Natural is
```

```
return The Bag. The Back;
      end Unique_Extent_Of;
     in
for Index in 1 .. In_The_Bag.The_Back loop
   if The_Item = In_The_Bag.The_Items(Index).The_Item then
        return In_The_Bag.The_Items(Index).The_Count;
   end if;
end loop;
raise Item_Is_Not_In_Bag;
      end Number Of:
      function Is_Empty (The_Bag : in Bag) return Boolean is
     return (The_Bag.The_Back = 0);
end Is_Empty;
      begin
     function Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag) return Boolean is
      begin
            in
for Index in 1 .. Of_The_Bag.The_Back loop
   if Of_The_Bag.The_Items(Index).The_Item = The_Item then
        return True;
   end if;
      end loop;
return False;
end Is_A_Member;
      in
for Left_Index in 1 .. Left.The_Back loop
   Right_Index := Right.The_Back;
   while Right_Index > 0 loop
    if Left.The_Items(Left_Index).The_Item =
        Right.The_Items(Right_Index).The_Item then
        exit;
      begin
                          else
                  else
    Right_Index := Right_Index - 1;
end if;
end loop;
if Right_Index = 0 then
                   return False;
elsif Left. The Ltems (Left_Index). The Count >
Right. The Ltems (Right_Index). The Count then
return False;
      end loop;
return True;
end Is_A_Subset;
      function Is_A_Proper_Subset (Left : in Bag;
    Right : in Bag) return Boolean is
    Total_Left_Count : Natural := 0;
    Total_Right_Count : Natural := 0;
    Right_Index : Natural;
      begin
             for Left_Index in 1 .. Left.The_Back loop
                   Left_Index in 1 .. Left.The_Back loop
Right_Index := Right.The_Back;
while Right_Index > 0 loop
    if Left.The_Items(Left_Index).The_Item =
        Right.The_Items(Right_Index).The_Item then
                         else
Right_Index := Right_Index - 1;
end if;
                   end it;
end loop;
if Right_Index = 0 then
    return False;
elsif Left.The_Items(Left_Index).The_Count >
                          Right.The_Items(Right_Index).The_Count then return False;
                   end if;
                   end loop;
             end loop;
for Index in 1 . Right.The_Back loop
   Total_Right_Count := Total_Right_Count +
        Right.The_Items(Index).The_Count;
             end loop;
if Left.The_Back < Right.The_Back then
             return True;
elsif Left.The_Back > Right.The_Back then
return False;
                   return (Total_Left_Count < Total_Right_Count);
      end if;
end Is_A_Proper_Subset;
end Bag_Simple_Sequential_Bounded_Managed_Noniterator;
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

PSDL

```
{\tt TYPE\ Bag\_Simple\_Sequential\_Bounded\_Managed\_Noniterator} \\ {\tt SPECIFICATION}
   GENERIC
      Item : PRIVATE_TYPE
   OPERATOR COPY
SPECIFICATION
      TNPIT
      From_The_Bag : Bag,
To_The_Bag : Bag
OUTFUT
         To_The_Bag : Bag
      EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
   OPERATOR Clear
SPECIFICATION
INPUT
         The_Bag : Bag
      OUTPUT
The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   OPERATOR Add
SPECIFICATION
      INPUT
         The_Item : Item,
To_The_Bag : Bag
      OUTPUT
To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   OPERATOR Remove
   SPECIFICATION
INPUT
The_Item : Item,
     From_The_Bag : Bag
OUTPUT
From_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Union
   SPECIFICATION
     PECIFICATION
INPUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   OPERATOR Intersection
   SPECIFICATION
INPUT
Of_The_Bag : Bag,
     And_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
     EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Difference SPECIFICATION
      TNPIT
        Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
     To_The_Bag : Bag
EXCEPTIONS
     OUTPUT
```

```
Overflow, Item_Is_Not_In_Bag
   OPERATOR IS_Equal SPECIFICATION
  SPECIFICATION
INFUT
Left: Bag,
Right: Bag
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
END
  OPERATOR Extent_Of
SPECIFICATION
INPUT
         NPUT
The_Bag : Bag
      OUTPUT

Result : Natural

EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Unique_Extent_Of SPECIFICATION
      INPUT
         The_Bag : Bag
      OUTPUT
Result : Natural
      EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_Empty
SPECIFICATION
      INPUT
The_Bag : Bag
OUTPUT
         Result : Boolean
     EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
  OPERATOR IS A Member SPECIFICATION INPUT The Item: Item, Of The Bag: Bag OUTPUT RESULT: BOOLEAN EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_A_Subset
SPECIFICATION
INPUT
Left : Bag,
      Right : Bag
OUTPUT
Result : Boolean
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR IS_A_Proper_Subset
SPECIFICATION
INPUT
Left : Bag,
Right : Bag
OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
IMPLEMENTATION ADA Bag_Simple_Sequential_Bounded_Managed_Noniterator
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
type Item is private;
package Bag_Simple_Sequential_Unbounded_Managed_Iterator is
      type Bag is limited private;
                                          (From_The_Bag : in
      procedure Copy
                                           To_The_Bag
(The_Bag
(The_Item
                                                                : in out Bag);
: in out Bag);
: in Item;
      procedure Clear
      procedure Add
                                                                   in out Bag);
                                            To The Bag
                                         (The_Item
From_The_Bag
(Of_The_Bag
      procedure Remove
                                                                   in
                                                                               Item:
                                                                   in out Bag);
      procedure Union
                                                                  in
                                                                            Bag:
     And The Bag
To The Bag
procedure Intersection (Of The Bag
                                                                   in
                                                                              Barr:
                                                                   in out Bag);
in Bag;
                                          And_The_Bag
And_The_Bag
To_The_Bag
(Of_The_Bag
And_The_Bag
To_The_Bag
                                                                   in
                                                                              Bag:
                                                                   in out Bag);
in Bag;
      procedure Difference
                                                                : in Bag;
: in Bag;
: in out Bag);
    modified by Tuan Nguyen and Vincent Hong
date: 7 April 1995
adding procedures to replace functions
                                                                       : in Bag;
: in Bag;
      procedure Is_Equal
                                                    Right
Result
(The_Bag
Result
                                                                         out Boolean);
in Bag;
out Natural);
in Bag;
      procedure Extent_Of
      procedure Unique_Extent_Of
                                                    (The_Bag
                                                                         in Bag;
out Natural);
in Bag;
out Boolean);
in Item;
                                                     Result
                                                    (The_Bag
Result
      procedure Is_Empty
      procedure Is_A_Member
                                                    (The Item
                                                     Of_The_Bag
                                                                       : in Bag;
```

```
out Boolean);
in Bag;
in Bag;
                                                Result
(Left
     procedure Is A Subset
                                                 Right
Result
                                                                    out Boolean);
in Bag;
     procedure Is_A_Proper_Subset
                                                 Right
                                                                   in Bag;
                                                 Result
                                                                 · out Boolean):
     end of modification
                                                               : in Bag;
: in Bag)
: in Bag)
      function Is_Equal
                                               (Left
                                                                               return Boolean;
return Natural;
return Natural;
                                              Right
(The_Bag
      function Extent_Of
function Unique_Extent_Of
function Number_Of
                                               (The_Bag : in Bag)
(The_Item : in Item;
In_The_Bag : in Bag)
                                              (The_Bag : in Bag)
(The_Item : in Item;
Of_The_Bag : in Bag)
     function Is_Empty
function Is_A_Member
                                                                               return Boolean;
                                                                             return Boolean;
                                                                 in Bag;
in Bag;
in Bag;
      function Is_A_Subset
                                               (Left
                                              Right
(Left
                                                                               return Boolean:
     function Is A Proper_Subset
                                                                             return Boolean;
                                               Right
                                                                : in Bag)
     generic
     Overflow : exception; Item_Is_Not_In_Bag : exception;
private
type Node;
  type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Managed_Iterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
                        "Restricted Rights Legend"
-- "Restricted Rights Legend"

-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body Bag_Simple_Sequential_Unbounded_Managed_Iterator is
       type Node is
               record
                      The_Item : Item;
The_Count : Positive;
Next : Bag;
               end record;
       procedure Free (The_Node : in out Node) is
begin
    The_Node.The_Count := 1;
       end Free;
       The_Node.Next := To_Next;
end Set_Next;
        function Next_Of (The_Node : in Node) return Bag is
       begin
              return The_Node.Next;
       end Next Of:
       Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
       in
Node_Manager.Free(To_The_Bag);
if From_The_Bag /= null then
To_The_Bag := Node_Manager.New_Item;
To_The_Bag := Node_Manager.New_Item;
To_The_Bag.The_Count := From_Index.The_Item;
To_The_Bag.The_Count := From_Index.The_Count;
To_Index := To_The_Bag;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := Node_Manager.New_Item;
To_Index := To_Index.Next;
To_Index.The_Item := From_Index.The_Item;
To_Index.The_Count := From_Index.The_Count;
From_Index := From_Index.Next;
end loop;
       begin
              end loop;
end if;
       exception
   when Storage_Error =>
    raise Overflow;
       procedure Clear (The_Bag : in out Bag) is begin
              Node_Manager.Free(The_Bag);
       end Clear;
       while Index /= null loop
if Index.The_Item = The_Item then
Index.The_Count := Index.The_Count + 1;
                             return;
                      else
                             Index := Index.Next;
               end if;
end loop;
               Temporary_Node := Node_Manager.New_Item;
               Temporary_Node The_Item := The_Item;
Temporary_Node The_Count := 1;
Temporary_Node Next := To_The_Bag;
               To_The_Bag := Temporary_Node;
       TO_INC___
exception
when Storage_Error =>
raise Overflow;
```

```
(The_Item : in Item;
From_The_Bag : in out Bag) is
      procedure Remove (The_Item
              Previous : Bag;
Index : Bag := From_The_Bag;
      begin
             Node_Manager.Free(Index);
                           else
                                  Previous.Next := Index.Next;
Index.Next := null;
Node_Manager.Free(Index);
                           end if:
                           return;
                    else
                          Previous := Index;
Index := Index.Next;
                    end if;
             end loop;
raise Item_Is_Not_In_Bag;
       end Remove:
      procedure Union (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
From_Index : Bag : Of_The_Bag;
To_Index : Bag;
             To Top
                                          Bag:
              Temporary_Node : Bag
      begin
            in
Node_Manager.Free(To_The_Bag);
while From_Index /= null loop
   Temporary_Node := Node_Manager.New_Item;
   Temporary_Node.The_Item := From_Index.The_Item;
   Temporary_Node.The_Count := From_Index.The_Count;
   Temporary_Node.Next := To_The_Bag;
   To_The_Bag := Temporary_Node;
   From_Index := From_Index.Next;
           To_Index := To_Index.Next;
end if;
                    end 1r;
end loop;
if To_Index = null then
   Temporary_Node := Node_Manager.New_Item;
   Temporary_Node.The_Item := From_Index.The_Item;
   Temporary_Node.The_Count := From_Index.The_Count;
                           Temporary_Node.Next := To_The_Bag;
To_The_Bag := Temporary_Node;
                    else
To_Index.The_Count :=
To_Index.The_Count + From_Index.The_Count;
            end 11;
From_Index := From_Index.Next;
end loop;
      exception
when Storage_Error =>
raise Overflow;
      end Union:
      procedure Intersection (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
Of_Index : Bag := Of_The_Bag;
And_Index : Bag;
Temporary_Node : Bag;
      begin
            Of Index. The Count:
                                  else
                                        Temporary_Node.The_Count :=
And Index. The Count;
                                  end if;
                                  Temporary_Node.Next := To_The_Bag;
To_The_Bag := Temporary_Node;
exit;
```

```
And_Index := And_Index.Next;
end if;
           end loop;
Of_Index := Of_Index.Next;
      end loop;
exception
when Storage_Error =>
raise Overflow;
end Intersection;
procedure Difference (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
Of_Index : Bag := Of_The_Bag;
      And_Index : Bag;
Temporary_Node : Bag;
begin
     in
Node_Manager.Free(To_The_Bag);
while Of_Index /= null loop
    And_Index := And_The_Bag;
    while And_Index /= null loop
        if Of_Index.The_Item = And_Index.The_Item then
                 else
                 And_Index := And_Index.Next; end if;
          And_In
Temporary_Node.Next := To_The_Bag;
To_The_Bag := Temporary_Node;
            end if:
      Of_Index := Of_Index.Next;
end loop;
 exception
exception
   when Storage_Error =>
      raise Overflow;
end Difference;
modified by Tuan Nguyen and Vincent Hong
date: 8 April 1995
adding procedures to replace functions
procedure Is_Equal (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
Result := Is_Equal(Left,Right);
end Is_Equal;
procedure Extent_Of (The_Bag : in Bag;
Result : out Natural) is
begin
      Result := Extent_Of(The_Bag);
 end Extent_Of;
Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;
procedure Number_Of (The_Item : in Item;
In_The_Bag : in Bag;
Result : out Positive) is
begin
           Result := Number_Of(The_Item,In_The_Bag);
 end Number_Of;
procedure Is_Empty (The_Bag : in Bag;
Result : out Boolean) is
Result := Is_Empty(The_Bag);
end Is_Empty;
procedure Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag;
Result : out Boolean) is
Result := Is_A_Member(The_Item,Of_The_Bag);
end Is_A_Member;
procedure Is_A_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
      Result := Is_A_Subset(Left,Right);
 end Is_A_Subset;
Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
end of modification
```

```
begin
      while Left_Index /= null loop
            Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
                        evit:
                 exit;
else
Right_Index := Right_Index.Next;
end if;
            return False;
elsif Left_Index.The_Count /= Right_Index.The_Count then
return False;
else
else

Left_Count := Left_Count + 1;

Left_Index := Left_Index.Next;

end if;
end loop;
Right_Index := Right;
while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
end loop;
return (Left_Count = Right_Count);
end Is_Equal;
function Extent_Of (The_Bag : in Bag) return Natural is
   Count : Natural := 0;
   Index : Bag := The_Bag;
begin
while Index /= null loop
Count := Count + Index.The_Count;
Index := Index.Next;
       end loop;
return Cou
end Extent_Of;
function Unique_Extent_Of (The_Bag : in Bag) return Natural is
   Count : Natural := 0;
   Index : Bag := The_Bag;
begin
      while Index /= null loop
Count := Count + 1;
Index := Index.Next;
end loop;
return Count;
end Unique_Extent_Of;
begin
      while Index /= null loop
   if The_Item = Index.The_Item then
      return Index.The_Count;
             else
      Index := Index.Next;
end if;
end loop;
raise Item_Is_Not_In_Bag;
end Number_Of;
function Is_Empty (The_Bag : in Bag) return Boolean is
        return (The_Bag = null);
end Is_Empty;
begin
      in
while Index /= null loop
   if The_Item = Index.The_Item then
        return True;
   end if;
   Index := Index.Next;
end loop;
return False;
end Is_A_Member;
function Is_A_Subset (Left : in Bag;
   Right : in Bag) return Boolean is
   Left_Index : Bag := Left;
   Right_Index : Bag;
Right_Index := Right_Index.Next;
end if;
end loop;
if Right_Index = null then
             return False;
elsif Left_Index.The_Count > Right_Index.The_Count then
return False;
else
                   Left_Index := Left_Index.Next;
             end if;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Managed_Iterator SPECIFICATION
  GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_Bag: Bag,
To_The_Bag: Bag
OUTPITE
      OUTPUT
To_The_Bag : Bag
EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   OPERATOR Clear
SPECIFICATION
      INPUT
      The_Bag : Bag
OUTPUT
The_Bag : Bag
      EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Add
SPECIFICATION
      INPUT
The_Item : Item,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
EXCEPTIONS
      INPUT
          Overflow, Item_Is_Not_In_Bag
   OPERATOR Remove
    SPECIFICATION
INPUT
          The_Item : Item,
          From The Bag : Bag
      OUTPUT
From_The_Bag : Bag
EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Union SPECIFICATION
       INPUT
          IPUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
      OUTPUT
To_The_Bag : Bag
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
    EMD
   OPERATOR Intersection SPECIFICATION
       INPUT
          Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag,
       OUTPUT
       OUTPUT
To_The_Bag : Bag
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
    END
    OPERATOR Difference
SPECIFICATION
      INPUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
       OUTPUT
       To_The_Bag : Bag
EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
    OPERATOR Is_Equal SPECIFICATION
       INPUT
```

```
Left : Bag,
Right : Bag
OUTPUT
        Result : Boolean
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Extent_Of SPECIFICATION
     INPUT
The_Bag : Bag
     OUTPUT
Result : Natural
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Unique_Extent_Of SPECIFICATION
     INPUT
The_Bag : Bag
OUTPUT
        Result : Natural
     EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR IS_Empty SPECIFICATION
     INPUT
        The_Bag : Bag
     OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   END
  OPERATOR Is_A_Member
SPECIFICATION
     INPUT
The_Item : Item,
Of_The_Bag : Bag
     OUTPUT

Result : Boolean

EXCEPTIONS

Overflow, Item_Is_Not_In_Bag
   END
  OPERATOR IS_A_Subset
SPECIFICATION
INPUT
Left : Bag,
Right : Bag
OUTPUT
        Result : Boolean
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_A_Proper_Subset
SPECIFICATION
INPUT
Left : Bag,
Right : Bag
OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   מאפ
   OPERATOR Iterate SPECIFICATION
GENERIC

GENERIC

Process: PROCEDURE[The_Item : in[t : Item], The_Count : in[t : Positive], Continue : out[t : Boolean]]

INPUT
      Over_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   END
```

IMPLEMENTATION ADA Bag_Simple_Sequential_Unbounded_Managed_Iterator

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
type Item is private;
package Bag_Simple_Sequential_Unbounded_Managed_Noniterator is
      type Bag is limited private;
                                       (From_The_Bag : To_The_Bag : (The_Bag : (The_Item : To_The_Bag :
                                                              in Bag;
in out Bag);
in out Bag);
     procedure Copy
     procedure Clear
procedure Add
                                                              in Item;
in out Bag);
                                       (The_Item
From_The_Bag
(Of_The_Bag
      procedure Remove
                                                              in
                                                                         Item:
                                                              in out Bag);
in Bag;
     procedure Union
                                         And The Bag
                                                              in
                                                                         Bag:
     TO_The_Bag
To_The_Bag
procedure Intersection (Of_The_Bag
And_The_Bag
                                                              in out Bag);
in Bag;
                                                              in
                                                                         Baq:
                                       To_The_Bag
(Of_The_Bag
And_The_Bag
                                                              in out Bag);
     procedure Difference
                                                           : in Bag;
: in out Bag);
                                         To_The_Bag
     modified by Tuan Nguyen and Vincent Hong date: 7 April 1995
     adding procedures to replace functions
                                                                  : in Bag;
: in Bag;
     procedure Is_Equal
                                                 Right
                                                Result
(The_Bag
Result
                                                                   out Boolean);
     procedure Extent_Of
                                                                    in Bag;
out Natural);
                                                (The Bag
                                                                   in Bag:
     procedure Unique_Extent_Of
                                                                   out Natural):
                                                Result
(The_Bag
     procedure Is_Empty
```

```
: out Boolean);
                                              Result
                                                               in Item;
in Bag;
out Boolean);
     procedure Is_A_Member
                                             (The_Item
Of_The_Bag
                                              Result
     procedure Is_A_Subset
                                             (Left
                                                               in Bag:
                                                               in Bag;
out Boolean);
in Bag;
                                              Result
                                             (Left
     procedure Is_A_Proper_Subset
                                              Right
Result
                                                               in Bag;
out Boolean);
-- end of modification
                                            (Left
                                                            : in Bag;
     function Is Equal
                                            Right
(The_Bag
(The_Bag
                                                           : in Bag)
: in Bag)
: in Bag)
: in Item;
                                                                          return Boolean;
     function Extent_Of
function Unique_Extent_Of
function Number_Of
                                                                          return Natural;
return Natural;
                                            (The Item
                                             In_The_Bag : in Bag)
                                                                          return
Positive;
                                                            in Bag)
                                                                          return Boolean;
     function Is_Empty
function Is_A_Member
                                            (The Bag
                                            (The_Item
Of_The_Bag
                                                              in Item;
in Bag)
                                                                          return Boolean;
     function Is_A_Subset
                                            (Left
                                                              in Bag;
                                                                          return Boolean;
     function Is_A_Proper_Subset (Left
                                                              in Bag;
                                                                          return Boolean:
                                                            : in Bag)
     Overflow
                               : exception;
     Item_Is_Not_In_Bag : exception;
type Node;
type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Managed_Noniterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
(C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
All Rights Reserved
-- Serial Number 0100219
                          "Restricted Rights Legend"
-- Westricted Rights Legend*
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body Bag_Simple_Sequential_Unbounded_Managed_Noniterator is
        type Node is
                       The_Item : Item;
The_Count : Positive;
                                            : Bag;
                end record;
       procedure Free (The_Node : in out Node) is begin
                The_Node.The_Count := 1;
       procedure Set_Next (The_Node : in out Node;
To_Next : in Bag)
       begin
                The Node.Next := To_Next;
        function Next_Of (The_Node : in Node) return Bag is
       return The_Node.Next;
end Next_Of;
       Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
       begin
               Node_Manager.Free(To_The_Bag);
if From_The_Bag /= null then
To_The_Bag := Node_Manager.New_Item;
To_The_Bag := Node_Manager.New_Item;
To_The_Bag.The_Item := From_Index.The_Item;
To_The_Bag.The_Count := From_Index.The_Count;
To_Index := To_The_Bag;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := Node_Manager.New_Item;
To_Index := To_Index.Next;
To_Index := To_Index.Next;
To_Index.The_Item := From_Index.The_Item;
To_Index.The_Item := From_Index.The_Count;
From_Index := From_Index.Next;
end loop;
                Node Manager Free (To The Bag);
                end loop;
end if;
        exception
        when Storage_Error =>
    raise Overflow;
end Copy;
        procedure Clear (The_Bag : in out Bag) is begin
                Node_Manager.Free(The_Bag);
        end Clear;
       begin
while Index /= null loop
if Index.The_Item = The_Item then
Index.The_Count := Index.The_Count + 1;
                        else
    Index := Index.Next;
end if;
                end if;
end loop;
Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Item := The_Item;
Temporary_Node.The_Count := 1;
Temporary_Node.Next := To_The_Bag;
To_The_Bag := Temporary_Node;
         exception when Storage_Error => raise Overflow;
```

```
cedure Remove (The_Item : in Item;
    From_The_Bag : in out Bag) is
Previous : Bag;
Index : Bag := From_The_Bag;
         procedure Remove (The_Item
      Index : pay
begin
while Index /= null loop
while Index.The_Item = The_Item then
    if Index.The_Count > 1 then
        Index.The_Count := Index.The_Count - 1;
    elsif Previous = null then
        From_The_Bag := From_The_Bag.Next;
        Index.Next := null;
        Node_Manager.Free(Index);
else
                                                Previous.Next := Index.Next;
Index.Next := null;
Node_Manager.Free(Index);
                                       end if;
                                      return;
                             Previous := Index;
Index := Index.Next;
end if;
                  end loop;
raise Item_Is_Not_In_Bag;
         end Remove:
        procedure Union (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
From_Index : Bag := Of_The_Bag;
To_Index : Bag;
Bag;
                   To Top
                   Temporary_Node : Bag;
                 in
Node_Manager.Free(To_The_Bag);
while From_Index /= null loop
   Temporary_Node := Node_Manager.New_Item;
   Temporary_Node.The_Item := From_Index.The_Item;
   Temporary_Node.The_Count := From_Index.The_Count;
   Temporary_Node.Next := To_The_Bag;
   To_The_Bag := Temporary_Node;
   From_Index := From_Index.Next;
end_loop:
                From_Index :- ...
end loop;
From_Index := And_The_Bag;
To_Top := To_The_Bag;
while From_Index /= null loop
To_Index := To_Top;
while To_Index /= null loop
if From_Index.The_Item = To_Index.The_Item then
exit;

alse

- To Index.Next;
                                       else
To_Index := To_Index.Next;
end if;
                              end loon:
                             end loop;
if To_Index = null then
   Temporary_Node := Node_Manager.New_Item;
   Temporary_Node.The_Item := From_Index.The_Item;
   Temporary_Node.The_Count := From_Index.The_Count;
   Temporary_Node.Next := To_The_Bag;
   To_The_Bag := Temporary_Node;
                             else
                             To_Index.The_Count := To_Index.The_Count + From_Index.The_Count; end if;
                             From_Index := From_Index.Next;
                   end loop;
         exception
when Storage_Error =>
raise Overflow;
        Of_Index
And_Index
                                                         : Bag;
                    Temporary_Node : Bag;
                   in
Node_Manager.Free(To_The_Bag);
while Of_Index /= null loop
And_Index := And_The_Bag;
while And_Index /= null loop
if Of_Index.The_Item = And_Index.The_Item then
    Temporary_Node := Node_Manager.New_Item;
    Temporary_Node.The_Item := Of_Index.The_Item;
if Of_Index.The_Count < And_Index.The_Count then
                                                             Temporary_Node.The_Count :=
Of_Index.The_Count;
                                                  else
   Temporary_Node.The_Count :=
And_Index.The_Count;
                                                   end if;
                                                  Temporary_Node.Next := To_The_Bag;
To_The_Bag := Temporary_Node;
```

```
else
    And_Index := And_Index.Next;
    end if;
    end loop;
    Of_Index := Of_Index.Next;
end loop;
eption
    exception
when Storage_Error =>
raise Overflow;
end Intersection;
    procedure Difference (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
Of_Index : Bag : Of_The_Bag;
And_Index : Bag;
          Temporary_Node : Bag;
          in
Node_Manager.Free(To_The_Bag);
while Of_Index /= null loop
And_Index := And_The_Bag;
while And_Index /= null loop
if Of_Index.The_Item = And_Index.The_Item then
                          exit:
                     else
                    And_Index := And_Index.Next;
end if;
              of_Index := Of_Index.Next;
end loop;
     exception
     exception
   when Storage_Error =>
       raise Overflow;
end Difference;
    modified by Tuan Nguyen and Vincent Hong
date: 8 April 1995
adding procedures to replace functions
                               (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
    procedure Is_Equal (Left
          Result := Is_Equal(Left,Right);
    end Is_Equal;
    procedure Extent_Of (The_Bag : in Bag;
Result : out Natural) is
          Result := Extent_Of(The_Bag);
    end Extent_Of;
    begin
    Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;
    procedure Number_Of (The_Item : in Item;
In_The_Bag : in Bag;
Result : out Positive) is
     begin
              Result := Number_Of(The_Item, In_The_Bag);
     end Number_Of;
    Result := Is_Empty(The_Bag);
    end Is_Empty;
    Result := Is_A_Member(The_Item,Of_The_Bag);
    end Is_A_Member;
    procedure Is_A_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
    Result := Is_A_Subset(Left,Right);
end Is_A_Subset;
    procedure Is_A_Proper_Subset (Left
                                             (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
    Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
-- end of modification
```

```
begin
      while Left_Index /= null loop
   Right_Index := Right;
   while Right_Index /= null loop
      if Left_Index.The_Item = Right_Index.The_Item then
                      exit:
               Right_Index := Right_Index.Next;
end if;
           end loop;
if Right_Index = null then
            return False;
elsif Left_Index.The_Count /= Right_Index.The_Count then
return False;
           Left_Count := Left_Count + 1;
   Left_Index := Left_Index.Next;
end if;
      end 1r;
end loop;
Right_Index := Right;
while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
end loop;
return (Left_Count = Right_Count);
end Is_Equal;
function Extent_Of (The_Bag : in Bag) return Natural is
   Count : Natural := 0;
   Index : Bag := The_Bag;
function Unique_Extent_Of (The_Bag : in Bag) return Natural is
   Count : Natural := 0;
   Index : Bag := The_Bag;
lndex : Bag := Ine_Be
begin
    while Index /= null loop
    Count := Count + 1;
    Index := Index.Next;
end loop;
    return Count;
 end Unique_Extent_Of;
Index := Index.Next;
end if;
      end loop;
raise Item_Is_Not_In_Bag;
 end Number_Of;
 function Is_Empty (The_Bag : in Bag) return Boolean is
begin
    return (The_Bag = null);
end Is_Empty;
Index : any ...
begin
while Index /= null loop
   if The_Item = Index.The_Item then
        return True;
   end if;
   Index := Index.Next;
and loop;
 end loop;
return False;
end Is_A_Member;
begin
      n
while Left_Index /= null loop
Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
                 exit;
                 Right_Index := Right_Index.Next;
end if;
           return False;
elsif Left_Index.The_Count > Right_Index.The_Count then
return False;
else
            end loop;
if Right_Index = null then
           Left_Index := Left_Index.Next; end if;
```

```
return False;
else
    Unique_Left_Count := Unique_Left_Count + 1;
    Total_Left_Count := Total_Left_Count + 1;
    Total_Left_Count := Total_Left_Count + 1;
    Left_Index := Left_Index.Next;
    end if;
    end loop;
    Right_Index := Right;
    while Right_Index /= null loop
        Unique_Right_Count := Unique_Right_Count + 1;
    Total_Right_Count := Total_Right_Count + 1;
    Right_Index := Right_Index.Next;
    end loop;
    if Unique_Left_Count < Unique_Right_Count then
        return True;
    elsif Unique_Left_Count > Unique_Right_Count then
        return False;
    else
        return (Total_Left_Count < Total_Right_Count);
    end if;
    end if,
    end Bag_Simple_Sequential_Unbounded_Managed_Noniterator;</pre>
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
   GENERIC

Item : PRIVATE_TYPE

OPERATOR Copy
   OPERATOR COPY
SPECIFICATION
INPUT
From_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Clear
SPECIFICATION
INPUT
         NPUT
The_Bag : Bag
     OUTPUT
The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   OPERATOR Add
   SPECIFICATION
      INPUT
        The_Item : Item,
To_The_Bag : Bag
      OUTPUT
To_The_Bag : Bag
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Remove
SPECIFICATION
      INPUT
      The_Item : Item,
From_The_Bag : Bag
OUTPUT
         From_The_Bag : Bag
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Union
   SPECIFICATION
     PECIFICATION
INPUT
Of The Bag : Bag,
And The Bag : Bag,
To The Bag : Bag
      OUTPUT
To_The_Bag : Bag
      EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Intersection SPECIFICATION
     INPUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
      OUTPUT
     OUT-01
To_The_Bag : Bag
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
  OPERATOR Difference SPECIFICATION
        Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
     OUTPUT
To_The_Bag : Bag
EXCEPTIONS
```

```
Overflow, Item_Is_Not_In_Bag
  EMD
   OPERATOR Is Equal
   SPECIFICATION
     INPUT
Left : Bag,
Right : Bag
     OUTFUT

Result : Boolean

EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Extent_Of
SPECIFICATION
INPUT
The_Bag : Bag
OUTPUT
     Result : Natural
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Unique_Extent_Of SPECIFICATION
     INPUT
     The_Bag : Bag
OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  END
  OPERATOR Is_Empty
SPECIFICATION
     INPUT
The_Bag : Bag
OUTPUT
     Result : Boolean
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
  END
  OPERATOR Is_A_Member
SPECIFICATION
    The_Item : Item,
Of_The_Bag : Bag
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_A_Subset
SPECIFICATION
INPUT
Left : Bag,
     Right : Bag
        Result : Boolean
     EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_A_Proper_Subset
SPECIFICATION
     INPUT
        Left : Bag,
Right : Bag
    RIGHT OUTPUT

RESULT : Boolean

EXCEPTIONS

Overflow, Item_Is_Not_In_Bag
IMPLEMENTATION ADA Bag_Simple_Sequential_Unbounded_Managed_Noniterator
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
generic
type Item is private;
package Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator is
         type Bag is limited private;
                                                       (From_The_Bag : in out Bag;
To_The_Bag : in out Bag);
(The_Bag : in out Bag);
(The_Item : in out Bag);
(The_Bag : in out Bag);
(The_Item : in out Bag);
        procedure Copy
        procedure Clear
procedure Add
                                                       To_The_bay
(The_Item
From_The_Bay
(Of_The_Bay
And_The_Bay
        procedure Remove
                                                                                       in out Bag);
in Bag;
in Bag;
        procedure Union
        To_The_Bag
procedure Intersection (Of_The_Bag
And_The_Bag
                                                                                      in out Bag);
in Bag;
in Bag;
                                                                                   : in out Bag);
: in Bag;
: in Bag;
: in out Bag);
: in Bag;
: in Bag;
: in Bag;
                                                        To_The_Bag
(Of_The_Bag
And_The_Bag
        procedure Difference
                                                         To_The_Bag
       modified by Tuan Nguyen and Vincent Hong
date: 7 April 1995
adding procedures to replace functions
                                                                                               in Bag;
in Bag;
out Boolean);
in Bag;
out Natural);
in Bag;
out Natural);
        procedure Is_Equal
                                                                      Right
                                                                   Right
Result
(The_Bag
Result
(The_Bag
Result
        procedure Unique_Extent_Of
                                                                    (The_Bag
Result
(The_Item
                                                                                               in Bag;
out Boolean);
in Item;
        procedure Is_Empty
        procedure Is_A_Member
                                                                      Of The Bag : in Bag;
```

```
Result
(Left
Right
Result
(Left
Right
                                                                 out Boolean);
in Bag;
     procedure Is A Subset
                                                                 in Bag;
out Boolean);
in Bag;
     procedure Is_A_Proper_Subset
                                                                  in Bag:
                                                               : out Boolean):
-- end of modification
                                                             : in Bag;
: in Bag)
: in Bag)
      function Is_Equal
                                             (Left
                                             Right
(The_Bag
                                                                             return Boolean;
return Natural;
      function Extent Of
     function Unique_Extent_Of
function Number_Of
                                             (The_Bag
(The_Item
In_The_Bag
                                                               in Bag)
in Item;
in Bag)
                                                                             return Natural:
Positive:
                                                             : in Bag)
: in Item;
: in Bag)
: in Bag;
: in Bag;
: in Bag;
: in Bag;
                                             (The_Bag : (The_Item : Of_The_Bag :
     function Is_Empty
function Is_A_Member
                                                                             return Boolean:
                                                                           return Boolean;
     function Is_A_Subset
                                             (Left
                                             Right
(Left
                                                                             return Boolean:
     function Is_A_Proper_Subset
                                                                             return Boolean;
                                              Right
     generic
     Overflow
                                : exception:
      Item_Is_Not_In_Bag : exception;
private
type Node;
  type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator is
      type Node is
           record
                 The_Item : Item;
The_Count : Positive;
Next : Bag;
           end record:
     From_Index : Bag := From_The_Bag;
To_Index : Bag;
           in
  if From_The_Bag = null then
    To_The_Bag := null;
           else
                 To_Index := To_The_Bag;
From_Index := From_Index.Next;
while From_Index /= null loop
   To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                                              The_Count =>
          Tn_Index := To_Index.Next;
From_Index := From_Index.Next;
end loop;
end if;
sptio-
From Index. The Count.
                                                                             => null);
      exception
when Storage_Error =>
raise Overflow;
      end Copy;
     procedure Clear (The_Bag : in out Bag) is begin
           The Bag := null;
           redure Add (The_Item : in Item;
To_The_Bag : in out Bag) is
Index : Bag := To_The_Bag;
      procedure Add (The_Item
      begin
           while Index /= null loop
if Index.The_Item = The_Item then
Index.The_Count := Index.The_Count + 1;
                       return:
                 else
Index := Index.Next;
end if;
           exception
    when Storage_Error =>
     raise Overflow;
end Add;
     procedure Remove (The_Item : in Item; From_The_Bag : in out Bag) is
           Previous : Bag;
                     : Bag := From_The_Bag;
      begin
           else
    Previous.Next := Index.Next;
end if;
                       return;
                  else
Previous := Index;
Index := Index.Next;
                  end if:
           end loop;
raise Item_Is_Not_In_Bag;
      end Remove:
```

```
procedure Union (Of_The_Bag : in
           And_The_Bag: in Bag;
To_The_Bag: in out Bag) is
From_Index: Bag := Of_The_Bag;
           To_Index : Bag;
To_Top : Bag;
           To_Top
          begin
                 From_Index := From_Index.Next;
           end loop:
          end loop;
From_Index := And_The_Bag;
To_Top := To_The_Bag;
while From_Index /= null loop
    To_Index := To_Top;
while To_Index /= null loop
    if From_Index.The_Item = To_Index.The_Item then
                            exit;
                      else
                      To_Index := To_Index.Next;
end if;
                end loop;
if To_Index = null then
To_The_Bag := new Node'(The_Item =>
From_Index.The_Item,
                                                        The_Count =>
From Index. The Count,
                                                        Next
                                                                      => To The Bag);
                      To_Index.The_Count :=
To_Index.The_Count + From_Index.The_Count;
                 end if:
                 From_Index := From_Index.Next;
           end loop;
     exception
when Storage_Error =>
raise Overflow;
     end Union:
     procedure Intersection (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
Of_Index : Bag := Of_The_Bag;
And_Index : Bag;
          in

To_The_Bag := null;
while Of_Index /= null loop

And_Index := And_The_Bag;
while And_Index /= null loop

if Of_Index.The_Item = And_Index.The_Item then

if Of_Index.The_Count < And_Index.The_Count then

To The Bag :=
     begin
                                 else
                      And_Index := And_Index.Next;
end if;
                end loop;
Of_Index := Of_Index.Next;
           end loop;
     exception
when Storage_Error =>
raise Overflow;
end Intersection;
     procedure Difference (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
Of_Index : Bag := Of_The_Bag;
And_Index : Bag;
     begin
           To_The_Bag:= null;
           r_Inc
exit;
else
                      And_Index := And_Index.Next;
end if;
                 end loop;
if And_Index = null then
                      To The Bag := new Node' (The Item =>
Of_Index.The_Item,
                                                        The_Count =>
Of_Index.The_Count,
```

```
Next
Next => To_The_Bag;
elsif Of_Index.The_Count > And_Index.The_Count then
To_The_Bag := new Node'(The_Item =>
Of_Index.The_Item,
                                                      The_Count =>
Of_Index.The_Count -
And Index. The Count.
                                                      Next
                                                                => To The Bag):
          end if;
    Of_Index := Of_Index.Next;
end loop;
     exception
when Storage_Error =>
raise Overflow;
     end Difference;
     modified by Tuan Nguyen and Vincent Hong
date: 8 April 1995
adding procedures to replace functions
     procedure Is_Equal (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
          Result := Is_Equal(Left,Right);
     end Is_Equal;
     begin
     Result := Extent_Of(The_Bag);
end Extent_Of;
     Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;
     Result := Number_Of(The_Item,In_The_Bag);
     end Number_Of;
     procedure Is_Empty (The_Bag : in Bag;
Result : out Boolean) is
     Result := Is_Empty(The_Bag);
end Is_Empty;
     procedure Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag;
Result : out Boolean) is
           Result := Is_A_Member(The_Item,Of_The_Bag);
     end Is_A_Member;
     procedure Is_A_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
           Result := Is_A_Subset(Left, Right);
     end Is_A_Subset;
     procedure Is_A_Proper_Subset (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
     Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
    end of modification
     begin
while Left_Index /= null loop
while Left_Index /= Right;
                Right Index := Right;
while Right Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
exit;
                      else
                           Right_Index := Right_Index.Next;
                end if;
end loop;
if Right_Index = null then
                return False;
elsif Left_Index.The_Count /= Right_Index.The_Count then
return False;
                Left_Count := Left_Count + 1;
Left_Index := Left_Index.Next;
end if;
     end if;
end loop;
Right_Index := Right;
while Right_Index /= null loop
    Right_Count := Right_Count + 1;
    Right_Index := Right_Index.Next;
end loop;
return (Left_Count = Right_Count);
end Is_Equal;
```

```
function Extent_Of (The_Bag : in Bag) return Natural is
         Count : Natural := 0;
Index : Bag := The_Bag;
    end Extent_Of;
     function Unique_Extent_Of (The_Bag : in Bag) return Natural is
         Count : Natural := 0;
Index : Bag := The_Bag;
    Index : Bag :- Index

begin

while Index /= null loop

Count := Count + 1;

Index := Index.Next;

end loop;

return Count;

end Unique_Extent_Of;
     begin while Index /= null loop if The_Item = Index.The_Item then Index.The_Count;
              return Index.The_Count;
else
   Index := Index.Next;
end if;
         end loop;
raise Item_Is_Not_In_Bag;
     end Number Of:
     function Is_Empty (The_Bag : in Bag) return Boolean is
    begin
    return (The_Bag = null);
end Is_Empty;
     Index : bag .
begin
while Index /= null loop
   if The_Item = Index.The_Item then
        return True;
   end if;
      Index := Index.Next;
    end loop;
return False;
end Is_A_Member;
    function Is_A_Subset (Left : in Bag;
    Right : in Bag) return Boolean is
    Left_Index : Bag := Left;
    Right_Index : Bag;
    Right_Index := Right_Index.Next;
end if;
              return False;
elsif Left_Index.The_Count > Right_Index.The_Count then
return False;
else
              Left_Index := Left_Index.Next;
end if;
    end loop;
return True;
end Is_A_Subset;
    Right_Index
    Right_Index := Right_Index.Next;
end if;
end loop;
if Right_Index = null then
              return False;
elsif Left_Index.The_Count > Right_Index.The_Count then
return False;
               else
                   Unique_Left_Count := Unique_Left_Count + 1;
Total_Left_Count := Total_Left_Count +
Left_Index.The_Count;
Left_Index := Left_Index.Next;
```

```
end loop;
   Right_Index := Right;
   while Right_Index /= null loop
        Unique_Right_Count := Unique_Right_Count + 1;
        Total_Right_Count := Total_Right_Count +
Right_Index.The_Count;
   Right_Index := Right_Index.Next;
   end loop;
   if Unique_Left_Count < Unique_Right_Count then
        return True;
   elsif Unique_Left_Count > Unique_Right_Count then
        return False;
   else
        return (Total_Left_Count < Total_Right_Count);
   end if;</pre>
```

```
end Is_A_Proper_Subset;

procedure Iterate (Over_The_Bag : in Bag) is
    The_Iterator : Bag := Over_The_Bag;
    Continue : Boolean;

begin
    while The_Iterator /= null loop
        Process(The_Iterator.The_Item, The_Iterator.The_Count,

Continue);
    exit when not Continue;
    The_Iterator := The_Iterator.Next;
    end loop;
    end Iterate;

end Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

66

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator SPECIFICATION
   GENERIC
   Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
       INPUT
         From_The_Bag : Bag,
To_The_Bag : Bag
      OUTPUT
       To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Clear
   SPECIFICATION
      INPUT
         The_Bag : Bag
      OUTPUT
The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Add
SPECIFICATION
      PECIFICATION
INPUT
The_Item : Item,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
      EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
   OPERATOR Remove
SPECIFICATION
      INPUT
         NPUT
The_Item : Item,
From_The_Bag : Bag
       OUTPUT
         JTPUT
From_The_Bag : Bag
      EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Union SPECIFICATION
      INPUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
       EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   OPERATOR Intersection
   SPECIFICATION
INPUT
Of_The_Bag : Bag,
      And_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
          To_The_Bag : Bag
       EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   OPERATOR Difference SPECIFICATION
      PECIFICATION
INFUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
   OPERATOR Is_Equal SPECIFICATION
```

INPUT

```
Left : Bag,
Right : Bag
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Extent_Of
  SPECIFICATION
INPUT
        The_Bag : Bag
     OUTPUT
        Result : Natural
     EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Unique_Extent_Of SPECIFICATION
     INPUT
        The_Bag : Bag
     OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_Empty
SPECIFICATION
     PECIFICATION
INPUT
The_Bag : Bag
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR IS_A_Member
SPECIFICATION
INPUT
The_Item : Item,
Of_The_Bag : Bag
OUTPUT
Result : Boolean
     EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_A_Subset
SPECIFICATION
INPUT
Left : Bag,
Right : Bag
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  OPERATOR Is_A_Proper_Subset SPECIFICATION
     INPUT
Left : Bag,
Right : Bag
     OUTPUT
     Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_Not_In_Bag
  END
  OPERATOR Iterate
SPECIFICATION
     GENERIC
Process: PROCEDURE[The_Item: in[t: Item], The_Count: in[t: Positive], Continue: out[t: Boolean]]
     INPUT
        Over The Bag : Bag
     EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
  END
IMPLEMENTATION ADA Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
generic
type Item is private;
package Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator is
      type Bag is limited private;
                                          (From_The_Bag : in Bag;
To_The_Bag : in out Bag);
(The_Bag : in out Bag);
     procedure Copy
     procedure Clear
procedure Add
                                          (The_Bag
(The_Item
To_The_Bag
(The_Item
                                                                              Item:
     procedure Remove
                                                                   in
                                          From_The_Bag
(Of_The_Bag
And_The_Bag
                                                                  in out Bag);
     procedure Union
                                                                              Bag;
                                          To_The_Bag
(Of_The_Bag
And_The_Bag
                                                                  in out Bag) :
                                                                  in
                                                                             Bag;
Bag;
     procedure Intersection
                                         To_The_Bag
(Of_The_Bag
And_The_Bag
To_The_Bag
                                                                  in out Bag);
     procedure Difference
                                                                  in
                                                                             Bag;
                                                               : in out Bag);
     modified by Tuan Nguyen and Vincent Hong
date: 7 April 1995
adding procedures to replace functions
                                                   (Left
                                                                         in Bag;
     procedure Is_Equal
                                                    Right
Result
                                                                      : in Bag;
: out Boolean);
                                                                        in Bag:
     procedure Extent_Of
                                                   (The_Bag
                                                                      : in Bag;
: out Natural);
: in Bag;
: out Natural);
                                                   Result
(The_Bag
Result
     procedure Unique_Extent_Of
     procedure Is_Empty
                                                   (The Bag
                                                                      : in Bag;
```

```
: out Boolean);
                                              Result
                                             (The_Item
Of_The_Bag
Result
     procedure Is_A_Member
                                                               in Item:
                                                                in Bag;
out Boolean);
in Bag;
     procedure Is_A_Subset
                                              Meft
                                              Right
                                                                in Bag;
out Boolean);
                                              Result
                                             (Left
                                                               in Bag;
     procedure Is_A_Proper_Subset
                                              Right
Result
                                                              : in Bag;
: out Boolean);
     end of modification
                                                            : in Bag;
: in Bag)
: in Bag)
: in Bag)
: in Item;
     function Is Equal
                                            (Left
                                            Right
(The_Bag
(The_Bag
(The_Item
                                                                          return Boolean:
     function Extent_Of
function Unique_Extent_Of
function Number_Of
                                                                           return Natural:
                                                                           return Natural;
                                             In_The_Bag
                                                            : in Bag)
                                                                           return
Positive;
                                                                          return Boolean;
                                            (The Bag
                                                              in Bag)
     function Is_Empty
function Is_A_Member
                                           (The_Item
Of_The_Bag
(Left
                                                              in Item;
                                                                           return Boolean;
     function Is_A_Subset
                                                              in Bag:
     Right function Is_A_Proper_Subset (Left
                                                              in Bag;
                                                                          return Boolean:
                                                                          return Boolean;
                                             Right
                                                            : in Bag)
     Overflow
                               : exception;
     Item_Is_Not_In_Bag : exception;
type Node;
  type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                  "Restricted Rights Legend"
   "Restricted Rights Legend" Use, duplication, or disclosure is subject to restrictions as set forth in subdivision (b) (3) (ii) of the rights in Technical Data and Computer Software Clause of FAR 52.227-7013. Manufacturer: Wizard software, 2171 S. Parfet Court, Lakewood, Colorado 80227 (1-303-987-1874)
package body Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator is
     type Node is
           record
                The_Item : Item;
The_Count : Positive;
Next : Bag;
           end record;
     begin
          if From_The_Bag = null then
    To_The_Bag := null;
          To_Index := To_The_Bag;
                Ac_invex := Trom_Index.Next;
while From_Index /= null loop
To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
From_Index.The_Count,
                                                            Next
                                                                          => null):
                      To_Index := To_Index.Next;
From_Index := From_Index.Next;
                 end loop;
           end if;
     exception
when Storage_Error =>
raise Overflow;
end Copy;
     procedure Clear (The_Bag : in out Bag) is
begin
    The_Bag := null;
end Clear;
     begin
           while Index /= null loop
   if Index.The_Item = The_Item then
        Index.The_Count := Index.The_Count + 1;
                      return:
                 else
           Index := Index.Next;
end if;
end loop;
           To_The_Bag := new Node'(The_Item => The_Item,
                                             The_Count => 1,
Next => To_The_Bag);
     exception
     when Storage_Error =>
    raise Overflow;
end Add;
     procedure Remove (The_Item : in Item; From_The_Bag : in out Bag) is
           Previous : Bag;
Index : Bag := From_The_Bag;
    end if;
                      return;
                 else
                 Previous := Index;
Index := Index.Next;
end if;
      end loop;
raise Item_Is_Not_In_Bag;
end Remove;
```

```
To_Top
                     : Bag;
        To_Index := To_Index.Next;
end if;
             end loop;
if To_Index = null then
To_The_Bag := new Node'(The_Item => From_Index.The_Item,
                                            The_Count =>
From Index. The Count.
                                                       => To The Bag);
                                            Next
             else
    To_Index.The_Count :=
    To_Index.The_Count + From_Index.The_Count;
    .
        end if;
From_Index := From_Index.Next;
end loop;
    exception
         when Storage_Error => raise Overflow;
    end Union;
    procedure Intersection (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
Of_Index : Bag := Of_The_Bag;
And_Index : Bag;
    begin
        else
                          end if;
                      exit;
                  else
                      And Index := And Index.Next;
             end if;
end loop;
Of_Index := Of_Index.Next;
         end loop;
    exception
when Storage_Error =>
    raise Overflow;
end Intersection;
    procedure Difference (Of_The_Bag : in Bag;
And_The_Bag : in Bag;
To_The_Bag : in out Bag) is
Of_Index : Bag := Of_The_Bag;
And_Index : Bag;
    begin
         in
To_The_Bag:= null;
while Of_Index /= null loop
And_Index := And_The_Bag;
while And_Index /= null loop
    if Of_Index.The_Item = And_Index.The_Item then
                 or_Ind
exit;
else
             else
And_Index := And_Index.Next;
end if;
end loop;
if And_Index = null then
                  To_The_Bag := new Node'(The_Item =>
Of Index.The Item
                                             The_Count =>
```

Of_Index.The_Count,

```
Of Index.The_Count -
And Index The Count.
                                                                  => To_The_Bag);
                                                      Next
                Of_Index := Of_Index.Next;
           end loop;
     exception
when Storage_Error =>
                raise Overflow;
      end Difference;
     modified by Tuan Nguyen and Vincent Hong
date: 8 April 1995
adding procedures to replace functions
     procedure Is_Equal (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
     begin
     Result := Is_Equal(Left,Right);
end Is_Equal;
     procedure Extent_Of (The_Bag : in Bag;
Result : out Natural) is
           Result := Extent_Of(The_Bag);
     end Extent_Of;
     procedure Unique_Extent_Of (The_Bag : in Bag;
Result : out Natural) is
     begin
     Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;
     procedure Number_Of (The_Item : in Item;
In_The_Bag : in Bag;
Result : out Positive) is
     begin
                Result := Number_Of(The_Item,In_The_Bag);
     end Number_Of;
     Result := Is_Empty(The_Bag);
      end Is_Empty;
     procedure Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag;
Result : out Boolean) is
           Result := Is_A_Member(The_Item,Of_The_Bag);
     end Is A Member;
     Result := Is_A_Subset(Left,Right);
end Is_A_Subset;
                                             (Left : in Bag;
Right : in Bag;
Result : out Boolean) is
     procedure Is_A_Proper_Subset (Left
     negin
  Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
     Right_Index . --..
begin
while Left_Index /= null loop
Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
exit;
                else
    Right_Index := Right_Index.Next;
end if;
end loop;
if Right_Index = null then
                return False;
elsif Left_Index.The_Count /= Right_Index.The_Count then
return False;
                else
          else
    Left_Count := Left_Count + 1;
Left_Index := Left_Index.Next;
end if;
end loop;
Right_Index := Right;
while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
end loop;
           end loop;
return (Left_Count = Right_Count);
      end Is_Equal;
```

```
function Extent Of (The Bag : in Bag) return Natural is
           Count : Natural := 0;
Index : Bag := The_Bag;
     begin
           while Index /= null loop
   Count := Count + Index.The_Count;
   Index := Index.Next;
     end loop;
return Count;
end Extent_Of;
     function Unique_Extent_Of (The_Bag : in Bag) return Natural is
   Count : Natural := 0;
   Index : Bag := The_Bag;
           while Index /= null loop
Count := Count + 1;
Index := Index.Next;
     end loop;
return Count;
end Unique_Extent_Of;
     Index : Bag := In_The_Bag;
     begin
           while Index /= null loop
if The_Item = Index.The_Item then
                 return Index.The_Count;
                       Index := Index.Next;
     end if;
end loop;
raise Item_Is_Not_In_Bag;
end Number_Of;
      function Is_Empty (The_Bag : in Bag) return Boolean is
           return (The_Bag = null);
     end Is_Empty;
     function Is_A_Member (The_Item : in Item;
Of_The_Bag : in Bag) return Boolean is
           Index : Bag := Of_The_Bag;
     begin
           while Index /= null loop
if The_Item = Index.The_Item then
                 return True;
end if;
Index := Index.Next;
     end loop;
return False;
end Is_A_Member;
     function Is_A_Subset (Left : in Bag;
    Right : in Bag) return Boolean is
    Left_Index : Bag := Left;
    Right_Index : Bag;
    Right_Index /= null loop

Right_Index /= null loop

Right_Index := Right;

while Right_Index /= null loop

if Left_Index.The_Item = Right_Index.The_Item then
                       else
                             Right_Index := Right_Index.Next;
                 end if;
end loop;
if Right_Index = null then
                 return False;
elsif Left_Index.The_Count > Right_Index.The_Count then
return False;
           Left_Index := Left_Index.Next;
end if;
end loop;
     return True;
end Is_A_Subset;
     begin
           while Left_Index /= null loop
Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
                             exit;
                 else
    Right_Index := Right_Index.Next;
    end if;
end loop;
if Right_Index = null then
    return False;
elsif Left_Index.The_Count > Right_Index.The_Count then
                 return False;
                       Unique_Left_Count := Unique_Left_Count + 1;
Total_Left_Count := Total_Left_Count +
```

end loop;
Right_Index := Right;
while Right_Index /= null loop
 Unique_Right_Count := Unique_Right_Count + 1;
 Total_Right_Count := Total_Right_Count +
Right_Index.The_Count;
 Right_Index := Right_Index.Next;
end loop;
if Unique_Left_Count < Unique_Right_Count then</pre>

return True;
elsif Unique_Left_Count > Unique_Right_Count then
 return False;
else
 return (Total_Left_Count < Total_Right_Count);
end if;
end Is_A_Proper_Subset;</pre>

end Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator;

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator SPECIFICATION
   GENERIC
   GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
         From_The_Bag : Bag,
To_The_Bag : Bag
      OUTPUT
      OUTPUT
To_The_Bag : Bag
EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Clear
   SPECIFICATION
INPUT
         The Bag : Bag
      OUTPUT
The_Bag : Bag
EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   OPERATOR Add
SPECIFICATION
      INPUT
         The_Item : Item,
To_The_Bag : Bag
      OUTPUT
To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
  OPERATOR Remove
SPECIFICATION
      INPUT
The_Item : Item,
From_The_Bag : Bag
OUTPUT
      OUTFUT
From_The_Bag : Bag
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
   OPERATOR Union SPECIFICATION
     INPUT
Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
      OUTPUT
      To_The_Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   END
  OPERATOR Intersection SPECIFICATION
     INPUT
Of The Bag : Bag,
And The Bag : Bag,
To The Bag : Bag
OUTPUT
To_The Bag : Bag
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   OPERATOR Difference
   SPECIFICATION
INPUT
Of_The_Bag : Bag,
     Of_The_Bag : Bag,
And_The_Bag : Bag,
To_The_Bag : Bag
OUTPUT
To_The_Bag : Bag
EXCEPTIONS
```

Overflow, Item_Is_Not_In_Bag

```
OPERATOR Is_Equal
   OPERATOR IS Equal
SPECIFICATION
INPUT
Left: Bag,
Right: Bag
OUTPUT
Result: Boolean
      EXCEPTIONS
          Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Extent_Of SPECIFICATION
  INPUT
The Bag : Bag
OUTPUT
Result : Natural
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
END
   OPERATOR Unique_Extent_Of
   SPECIFICATION
         The_Bag : Bag
      OUTPUT
Result : Natural
EXCEPTIONS
  EACEFFIONS
Overflow, Item_Is_Not_In_Bag
END
  OPERATOR IS_Empty
SPECIFICATION
INFUT
The_Bag : Bag
OUTPUT
Result : Boolean
EXCEPTIONS
OUTPICTOR ITEM IS
         Overflow, Item_Is_Not_In_Bag
   END
   OPERATOR Is_A_Member SPECIFICATION
      INPUT
      The_Item : Item,
Of_The_Bag : Bag
OUTPUT
      Result : Boolean
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
   OPERATOR Is_A_Subset SPECIFICATION
      INPUT
     Left : Bag,
Right : Bag
OUTPUT
      Result : Boolean
EXCEPTIONS
         Overflow, Item_Is_Not_In_Bag
   OPERATOR Is_A_Proper_Subset
SPECIFICATION
      INPUT
        Left : Bag,
Right : Bag
     OUTFUT
Result: Boolean
EXCEPTIONS
Overflow, Item_Is_Not_In_Bag
   END
IMPLEMENTATION ADA
Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator
```

LIST OB.13 SPECIFICATIONS

LISTS PROFILE CODES

OPERATORS	SIGNATURES	PROFILE CODES
COPY	A B -> B	3211
CLEAR	A -> A	2201
CONSTRUCT	A B -> B	3211
SET_HEAD	A B -> A	3211
IS_EQUAL	A B -> C	330
LENGTH_OF	A -> B	220
IS_NULL	A -> B	220
HEAD OF	A -> B	220
TAIL_OF	A -> B	220
PREDECESSOR_OF	A ->B	220

SET OF PROFILE: {3211,2201,330,220}

LIST DOUBLE BOUNDED MANAGED

```
Result : out Boolean);

(The_List : in List;

Result : out Item);

procedure Tail_Of (The_List : in List;

Result : out List);

procedure Predecessor_Of (The_List : in List);

procedure Predecessor_Of (The_List : in List);

-- end of modification

function Is_Equal (Left : in List) return Boolean;

function Length_Of (The_List : in List) return Natural;

function Is_Null (The_List : in List) return Natural;

function Head_Of (The_List : in List) return Boolean;

function Tail_Of (The_List : in List) return List;

function Predecessor_Of (The_List : in List) return List;

Overflow : exception;

Not_AL_Head : exception;

Private

type_List is

record

The_Head : Natural := 0;
end record;

Null_List : constant List := List'(The_Head => 0);
end List_Double_Bounded_Managed;
```

LIST DOUBLE BOUNDED MANAGED

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                             "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body List_Double_Bounded_Managed is
         type Node is
                 Previous : List;
The_Item : Item;
Mext : List;
        Heap : array(Positive range 1 .. The_Size) of Node;
         Free List : List:
        procedure Free (The_List : in out List) is
    Temporary_Node : List;
                 in
while The_List /= Null_List loop
   Temporary_Node := The_List;
   The_List := Heap(The_List.The_Head).Next;
   Heap(Temporary_Node.The_Head).Previous := Null_List;
   Heap(Temporary_Node.The_Head).Next := Free_List;
   Free_List := Temporary_Node;
                  end loop;
         end Free:
         function New_Item return List is
   Temporary_Node : List;
         begin
                  if Free_List = Null_List then
                           raise Storage_Error;
                 else
    Temporary_Node := Free_List;
    Free_List := Heap(Free_List.The_Head).Next;
    Heap(Temporary_Node.The_Head).Next := Null_List;
    return Temporary_Node;
end if;
New Temporary_Node;
         end New Item:
        To_Index
                in
Free(To_The_List);
From_The_List /= Null_List then
To_The_List := New_Item;
Heap(To_The_List.The_Head).The_Item :=
Heap(From_Index.The_Head).The_Item;
To_Index := To_The_List;
From_Index := Heap(From_Index.The_Head).Next;
while From_Index /= Null_List loop
Heap(To_Index.The_Head).Next := New_Item;
Heap(Heap(To_Index.The_Head).Next.The_Head).Previous
:=
                                    To_Index;
To_Index := Heap(To_Index.The_Head).Next;
                                    Heap(To_Index.The_Head).The_Item :=
    Heap(From_Index.The_Head).The_Item;
From_Index := Heap(From_Index.The_Head).Next;
                           end loop;
                  end if;
        exception when Storage_Error =>
                        raise Overflow;
        procedure Clear (The_List : in out List) is begin
        Free(The_List);
end Clear;
        begin
                 in
  if And_The_List = Null_List then
  And_The_List := New_Item;
  Heap(And_The_List.The_Head).The_Item := The_Item;
  elsif Heap(And_The_List.The_Head).Previous = Null_List then
  Temporary_Node := New_Item;
  Heap(Temporary_Node.The_Head).The_Item := The_Item;
  Heap(Temporary_Node.The_Head).Next := And_The_List;
  Heap(And_The_List.The_Head).Previous := Temporary_Node;
  And_The_List := Temporary_Node;
```

```
else
            raise Not_At_Head;
end if;
     exception
            when Storage_Error => raise Overflow;
     end Construct:
     Heap(Of_The_List.The_Head).The_Item := To_The_Item;
     exception
when Constraint_Error =>
     raise List_Is_Null;
end Set_Head;
     begin
           in
  if And_The_List = Null_List then
  if Heap(Of_The_List.The_Head).Next /= Null_List then
    Temporary_Node := Heap(Of_The_List.The_Head).Next;
    Heap(Temporary_Node.The_Head).Previous := Null_List;
    Heap(Of_The_List.The_Head).Next := Null_List;
                        And_The_List := Temporary_Node;
           And_The_List := Temporary_Node;
end if;
elsif Heap(And_The_List.The_Head).Previous = Null_List then
if Heap(Of_The_List.The_Head).Next /= Null_List then
Temporary_Node := Heap(Of_The_List.The_Head).Next;
Heap(Temporary_Node.The_Head).Previous := Null_List;
Heap(Of_The_List.The_Head).Previous := Null_List;
Heap(And_The_List.The_Head).Previous := Of_The_List;
And_The_List := Temporary_Node;
                  else
                        ;
Heap(And_The_List.The_Head).Previous := Of_The_List;
Heap(Of_The_List.The_Head).Next := And_The_List;
And_The_List := Null_List;
           end if;
else
           raise Not_At_Head;
end if;
     exception
when Constraint_Error =>
raise List_Is_Null;
end Swap_Tail;
   modified by Vincent Hong and Tuan Nguyen
date: 9 April 1995
adding procedures to replace functions
     procedure Is_Equal
                                            (Left
                                                          : in List:
                                              Right : in List;
Result : out Boolean) is
     Result := Is_Equal end Is_Equal;
                                            (Left.Right):
                                            (The_List : in List;
Result : out Natural) is
     procedure Length_Of
     begin
                                                  (The_List);
            Result := Length_Of
     end Length_Of;
                                            (The_List : in List;
Result : out Boolean) is
     procedure Is Null
                                                  (The_List);
           Result := Is_Null
     end Is_Null;
                                            (The_List : in List;
Result : out Item) is
     procedure Head_Of
     begin
     Result := Head_Of
end Head_Of;
                                                  (The_List);
                                            (The_List : in List;
Result : out List) is
     procedure Tail_Of
     begin
            Result := Tail_Of
                                                  (The List):
     end Tail_Of;
     Result := Predecessor_Of (The_List);
end Predecessor_Of;
-- end of modification
     begin
            while Left_Index /= Null_List loop
   if Heap(Left_Index.The_Head).The_Item /=
```

```
Heap(Right_Index.The_Head).The_Item then
return False;
end if;
Left_Index := Heap(Left_Index.The_Head).Next;
Right_Index := Heap(Right_Index.The_Head).Next;
end loop;
return (Right_Index = Null_List);
exception
when Constraint_Error =>
return False;
end Is_Equal;

function Length_Of (The_List : in List) return Natural is
Count : Natural := 0;
Index : List := The_List;

begin
while Index /= Null_List loop
Count := Count + 1;
Index := Heap(Index.The_Head).Next;
end loop;
return Count;
end Length_Of;

function Is_Null (The_List : in List) return Boolean is
begin
return (The_List = Null_List);
end Is_Null;

function Head_Of (The_List : in List) return Item is
begin
```

LIST DOUBLE BOUNDED MANAGED

PSDL

```
TYPE List_Double_Bounded_Managed
SPECIFICATION
GENERIC
Item : PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_List : List,
To_The_List : List
OUTPUT
To_The_List : List
EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
END
TYPE List_Double_Bounded_Managed
    OPERATOR Clear
SPECIFICATION
INPUT
The_List : List
        OUTPUT
The_List : List
        EXCEPTIONS
            Overflow, List_Is_Null, Not_At_Head
     END
    OPERATOR Construct
SPECIFICATION
        INPUT
The_Item : Item,
And_The_List : List
        OUTPUT
And_The_List : List
EXCEPTIONS
             Overflow, List_Is_Null, Not_At_Head
   OPERATOR Set_Head
SPECIFICATION
INPUT
Of_The_List : List,
To_The_Item : Item
OUTPUT
Of_The_List : List
EXCEPTIONS
            Overflow, List_Is_Null, Not_At_Head
    EMI
    OPERATOR SWap_Tail
SPECIFICATION
INPUT
Of_The_List : List,
And_The_List : List
OUTPUT
        OUTPUT
Of The List: List,
And The List: List
EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
    מעים
    OPERATOR Is_Equal
SPECIFICATION
INPUT
```

```
Left : List,
Right : List
OUTPUT
Result : Boolean
      EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
   END
   OPERATOR Length_Of SPECIFICATION
     PECIFICATION
INPUT
The_List : List
OUTPUT
Result : Natural
EXCEPTIONS
         Overflow, List_Is_Null, Not_At_Head
  OPERATOR Is_Null
SPECIFICATION
INPUT
The_List : List
OUTPUT
Result : Boolean
  Overflow, List_Is_Null, Not_At_Head END
     EXCEPTIONS
   OPERATOR Head_Of
   SPECIFICATION
     INPUT
The_List : List
OUTPUT
     Result : Item
EXCEPTIONS
        Overflow, List_Is_Null, Not_At_Head
   EMD
  OPERATOR Tail_Of
SPECIFICATION
INPUT
The_List : List
OUTPUT
Result : List
     EXCEPTIONS
        Overflow, List_Is_Null, Not_At_Head
  OPERATOR Predecessor_Of
SPECIFICATION
INPUT
The_List : List
     OUTPUT
     Result : List
EXCEPTIONS
        Overflow, List_Is_Null, Not_At_Head
   END
IMPLEMENTATION ADA List_Double_Bounded_Managed
```

LIST DOUBLE UNBOUNDED MANAGED

```
procedure Head_Of (The_List : in List;
Result : out Item);
Procedure Tail_Of (The_List : in List;
Result : out List);
Procedure Predecessor_Of (The_List : in List;
Result : out List);

-- end of modification

function Is_Equal (Left : in List) return Boolean;
function Length_Of (The_List : in List) return Boolean;
function Is_Null (The_List : in List) return Boolean;
function Head_Of (The_List : in List) return Boolean;
function Tail_Of (The_List : in List) return Boolean;
function Predecessor_Of (The_List : in List) return Item;
function Predecessor_Of (The_List : in List) return List;

Overflow : exception;
List_Is_Null : exception;
Not_At_Head : exception;

private
    type Node;
    type List is access Node;
    Null_List : constant List := null;
end List_Double_Unbounded_Managed;
```

LIST DOUBLE UNBOUNDED MANAGED

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FRR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body List_Double_Unbounded_Managed is
         type Node is
                  record
                           Previous : List;
The_Item : Item;
Next : List;
                  end record;
         procedure Free (The_Node : in out Node) is
         begin
                  The Node Previous := null:
         procedure Set_Next (The_Node : in out Node;
To_Next : in List)
                  The_Node.Next := To_Next;
         function Next Of (The Node : in Node) return List is
         begin
return The_Node.Next;
end Next_Of;
         package Node_Manager is new Storage_Manager_Sequential
                                              ager is new Storage_Manag
(Item Pointer
Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
        begin
                in
  Node_Manager.Free(To_The_List);
  if From_The_List /= null then
   To_The_List := Node_Manager.New_Item;
  To_The_List.The_Item := From_Index.The_Item;
  To_Index := To_The_List;
  From_Index := From_Index.Next;
  while From_Index /= null loop
   To_Index.Next := Node_Manager.New_Item;
   To_The Nort Persion_Index.
                                   To_Index.Next.Previous := To_Index;
To_Index := To_Index.Next;
To_Index.The_Item := From_Index.The_Item;
From_Index := From_Index.Next;
                 end loop;
end if;
         exception
   when Storage_Error =>
   raise Overflow;
         end Copy;
         procedure Clear (The_List : in out List) is
begin
   Node_Manager.Free(The_List);
         end Clear:
        procedure Construct (The_Item : in Item;
And_The_List : in out List) is
Temporary_Node : List;
                in
  if And_The_List = null then
   And_The_List := Node_Manager.New_Item;
  And_The_List.The_Item := The_Item;
elsif And_The_List.Previous = null then
  Temporary_Node := Node_Manager.New_Item;
  Temporary_Node.Next := And_The_List;
  And_The_List.Previous := Temporary_Node;
  And_The_List := Temporary_Node;
  And_The_List := Temporary_Node;
else
                  else
                 raise Not_At_Head;
end if;
         exception
         when Storage_Error =>
    raise Overflow;
end Construct;
         procedure Set_Head (Of_The_List : in out List;
```

```
To_The_Item : in
                                                                                  Item) is
               Of_The_List.The_Item := To_The_Item;
        exception
when Constraint_Error =>
raise List_Is_Null;
        end Set Head:
       procedure Swap_Tail (Of_The_List : in out List;
And_The_List : in out List) is
Temporary_Node : List;
              in
  if And_The_List = null then
  if Of_The_List.Next /= null then
    Temporary_Node := Of_The_List.Next;
    Temporary_Node.Previous := null;
    Of_The_List.Next := null;
    And_The_List := Temporary_Node;
              And_The_List := Temporary_Node;
end if;
elsif And_The_List.Previous = null then
if Of_The_List.Next /= null then
Temporary_Node := Of_The_List.Next;
Of_The_List.Next := And_The_List.Next;
Of_The_List.Next := And_The_List;
And_The_List.Previous := Of_The_List;
And_The_List.Previous := Of_The_List;
And_The_List := Temporary_Node;
                     And_The_List.Previous := Of_The_List;
   Of_The_List.Next := And_The_List;
   And_The_List := null;
end if;
               else
              raise Not_At_Head; end if;
       exception
when Constraint_Error =>
raise List_Is_Null;
end Swap_Tail;
      modified by Vincent Hong and Tuan Nguyen
date: 9 April 1995
adding procedures to replace functions
                                                       (Left : in List;
Right : in List;
Result : out Boolean) is
                                                     (Left
       procedure Is_Equal
      begin
               Result := Is_Equal (Left, Right);
       end Is_Equal;
                                                     (The_List : in List;
Result : out Natural) is
      procedure Length_Of
      Result := Length_Of
end Length_Of;
                                                             (The_List);
                                                     (The_List : in List;
Result : out Boolean) is
       procedure Is_Null
                                                            (The List):
       Result := Is_Null
end Is_Null;
                                                     (The_List : in List;
Result : out Item) is
      procedure Head_Of
      begin
      Result := Head_Of
end Head_Of;
                                                            (The_List);
                                                    (The_List : in List;
Result : out List) is
      procedure Tail_Of
      Result := Tail_Of
end Tail_Of;
                                                            (The_List);
      Result := Predecessor_Of (The_List);
       end Predecessor Of;
-- end of modification
      begin
             in
while Left_Index /= null loop
   if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
   end if;
   Left_Index := Left_Index.Next;
   Right_Index := Right_Index.Next;
end loop;
   return (Right_Index = null);
       exception when Constraint_Error =>
                      return False;
```

```
end Is Equal:
function Length_Of (The_List : in List) return Natural is
   Count : Natural := 0;
   Index : List := The_List;
Index : List := The_Li
begin
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Length_Of;
function Is_Null (The_List : in List) return Boolean is
begin
    return (The_List = null);
end Is_Null;
function Head_Of (The_List : in List) return Item is
begin
  return The_List.The_Item;
exception
```

```
when Constraint_Error =>
    raise List_Is_Null;
end Head_Of;
         function Tail_Of (The_List : in List) return List is
        runction Tail_Of (The_List :
begin
    return The_List.Next;
exception
    when Constraint_Error =>
        raise List_Is_Null;
end Tail_Of;
       function Predecessor_Of (The_List : in List) return List is
begin
    return The_List.Previous;
exception
    when Constraint_Error =>
        raise List_Is_Null;
end Predecessor_Of;
end List_Double_Unbounded_Managed;
```

LIST DOUBLE UNBOUNDED MANAGED

PSDL

```
TYPE List_Double_Unbounded_Managed
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
   Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INFUT
From_The_List: List,
To_The_List: List
OUTPUT
To_The_List: List
EVERPTIONS
       EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
   OPERATOR Clear
SPECIFICATION
      INPUT
The_List : List
      OUTPUT
The_List : List
       EXCEPTIONS
          Overflow, List_Is_Null, Not_At_Head
   OPERATOR Construct
SPECIFICATION
          The_Item : Item,
      And_The_List : List
OUTPUT
And_The_List : List
EXCEPTIONS
          Overflow, List_Is_Null, Not_At_Head
   OPERATOR Set_Head
SPECIFICATION
INFUT
Of_The_List : List,
To_The_Item : Item
OUTPUT
          Of_The_List : List
       EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
  OPERATOR Swap_Tail
SPECIFICATION
INPUT
Of The List : List,
And_The_List : List
OUTPUT
Of The List : List,
And The_List : List,
EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
END
   OPERATOR Is_Equal SPECIFICATION
       INPUT
```

```
Left : List,
Right : List
OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
  OPERATOR Length_Of
SPECIFICATION
INPUT
The_List : List
OUTPUT
Result : Natural
      EXCEPTIONS
         Overflow, List_Is_Null, Not_At_Head
  OPERATOR IS_Null
SPECIFICATION
INPUT
The_List : List
OUTPUT
Result : Boolean
EXCEPTIONS
OUTPTIONS
OUTPTION List IS
        Overflow, List_Is_Null, Not_At_Head
  END
  OPERATOR Head_Of
   SPECIFICATION
INPUT
The_List : List
     OUTPUT
Result : Item
EXCEPTIONS
         Overflow, List_Is_Null, Not_At_Head
  OPERATOR Tail_Of
   SPECIFICATION
INPUT
The_List : List
     OUTPUT
      Result : List
EXCEPTIONS
        Overflow, List_Is_Null, Not_At_Head
  OPERATOR Predecessor_Of SPECIFICATION
     INPUT
The_List : List
OUTPUT
Result : List
      EXCEPTIONS
         Overflow, List_Is_Null, Not_At_Head
  END
IMPLEMENTATION ADA List_Double_Unbounded_Managed
```

LIST DOUBLE UNBOUNDED UNMANAGED

```
procedure Head_Of (The_List : in List;
Result : out Item);
procedure Tail_Of (The_List : in List;
Result : out List);
procedure Predecessor_Of (The_List : in List);
procedure Predecessor_Of (The_List : in List);

-- end of modification

function Is_Equal (Left : in List);
Right : in List) return Boolean;
function Length_Of (The_List : in List) return Natural;
function Is_Null (The_List : in List) return Boolean;
function Head_Of (The_List : in List) return Education;
function Tail_Of (The_List : in List) return List;
function Predecessor_Of (The_List : in List) return List;

Overflow : exception;
List_Is_Null : exception;
Not_At_Head : exception;
private
type Node;
type List is access Node;
Null_List : constant List := null;
end List_Double_Unbounded_Unmanaged;
```

LIST DOUBLE UNBOUNDED UNMANAGED

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FRR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body List_Double_Unbounded_Unmanaged is
      type Node is
            record
                   Previous : List;
The_Item : Item;
Next : List;
                  Next
             end record;
      To_The_List : in out :
From_Index : List := From_The_List;
             To_Index : List;
      begin
   if From_The_List = null then
        To_The_List := null;
            else
To_The_List := new Node'(Previous => null,
The_Item => From_Index.The_Item,
Next => null);
                  To_Index := To_The_List;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := new Node'(Previous => To_Index,
                                                                   The_Item =>
From_Index.The_Item,
                                                                                 => null):
                        To_Index := To_Index.Next;
From_Index := From_Index.Next;
            end loop;
end if;
      exception
when Storage_Error =>
raise Overflow;
      end Copy:
      procedure Clear (The_List : in out List) is begin
            The List := null:
      end Clear;
                                      (The_Item : in Item;
And_The_List : in out List) is
     procedure Construct (The_Item
      begin
            if And_The_List = null then
           else
            raise Not_At_Head;
end if;
      exception
      when Storage_Error =>
    raise Overflow;
end Construct;
     Item) is
      begin
            Of_The_List.The_Item := To_The_Item;
      exception
when Constraint_Error =>
raise List_Is_Null;
     Temporary_nobegin
if And_The_List = null then
if Of_The_List.Next /= null then
Temporary_Node := Of_The_List.Next;
Temporary_Node.Previous := null;
Of_The_List.Next := null;
And_The_List := Temporary_Node;
            end if;
elsif And The_List.Previous = null then
if Of_The_List.Next /= null then
Temporary_Node := Of_The_List.Next;
```

```
Temporary_Node.Previous := null;
Of_The_List.Next := And_The_List;
And_The_List.Previous := Of_The_List;
And_The_List := Temporary_Node;
              else
And_The_List.Previous := Of_The_List;
Of_The_List.Next := And_The_List;
And_The_List := null;
end if;
          else
          raise Not_At_Head;
end if;
     exception
when Constraint_Error =>
     raise List_Is_Null;
end Swap_Tail;
-- modified by Vincent Hong and Tuan Nguyen
-- date: 9 April 1995
-- adding procedures to replace functions
                                      (Left : in List;
Right : in List;
Result : out Boolean) is
    procedure Is Equal
                                     (Left
     begin
          Result := Is_Equal (Left,Right);
     end Is_Equal;
                                     (The_List : in List;
    procedure Length_Of
                                      Result : out Natural) is
          Result := Length_Of
                                          (The List):
     end Length_Of;
                                     (The_List : in List;
Result : out Boolean) is
    procedure Is_Null
    begin
    Result := Is_Null
end Is_Null;
                                          (The List):
    procedure Head_Of
                                     (The_List : in List;
Result : out Item) is
          Result := Head Of
                                          (The_List);
    end Head_Of;
                                     (The_List : in List;
Result : out List) is
    procedure Tail Of
    Result := Tail_Of
end Tail_Of;
                                          (The_List);
    Result := Predecessor_Of (The_List);
end Predecessor_Of;
-- end of modification
    begin
         in
while Left_Index /= null loop
   if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
end if;
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
end loop;
         end loop;
return (Right_Index = null);
    exception
         when Constraint_Error =>
    return False;
end Is_Equal;
    begin
         while Index /= null loop
Count := Count + 1;
Index := Index.Next;
          end loop;
    return Con
end Length_Of;
    function Is_Null (The_List : in List) return Boolean is
    begin
    return (The_List = null);
end Is_Null;
     function Head_Of (The_List : in List) return Item is
    begin
         return The_List.The_Item;
```

```
when Constraint_Error =>
    raise List_Is_Null;
end Head_Of;
function Tail_Of (The_List : in List) return List is
begin
    return The_List.Next;
exception
    when Constraint_Error =>
        raise List_Is_Null;
end Tail_Of;
```

```
function Predecessor_Of (The_List : in List) return List is
begin
    return The_List.Previous;
exception
    when Constraint_Error =>
        raise List_Is_Null;
end Predecessor_Of;
end List_Double_Unbounded_Unmanaged;
```

LIST DOUBLE UNBOUNDED UNMANAGED

PSDL

```
TYPE List_Double_Unbounded_Unmanaged SPECIFICATION
   GENERIC
Item : PRIVATE_TYPE
    OPERATOR Copy
SPECIFICATION
       INPUT
      INPUT
From_The_List : List,
To_The_List : List
OUTPUT
To_The_List : List
EXCEPTIONS
          Overflow, List_Is_Null, Not_At_Head
    END
    OPERATOR Clear
    SPECIFICATION
INPUT
The_List : List
      OUTPUT
The_List : List
EXCEPTIONS
          Overflow, List_Is_Null, Not_At_Head
    OPERATOR Construct
SPECIFICATION
       INPUT
          The_Item : Item,
And_The_List : List
       OUTPUT
       OUTPUT
And_The_List : List
EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
    EMD
    OPERATOR Set_Head
SPECIFICATION
      PECIFICATION
INPUT
Of The List : List,
To The Item : Item
      TO_INE_Item: Item
OUTPUT
Of_The_List: List
EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
    OPERATOR Swap_Tail
    SPECIFICATION
INPUT
Of_The_List : List,
      Of The List : List
OUTPUT
Of The List : List
And The List : List
EXCEPTIONS
          Overflow, List_Is_Null, Not_At_Head
    OPERATOR IS_Equal
SPECIFICATION
INPUT
```

```
Left : List.
    Right : List
OUTPUT
Result : Boolean
    EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
  OPERATOR Length_Of SPECIFICATION
    INPUT
The_List : List
OUTPUT
       Result : Natural
    EXCEPTIONS
Overflow, List_Is_Null, Not_At_Head
  END
  OPERATOR IS_Null SPECIFICATION
    INPUT
The_List : List
OUTPUT
    Result : Boolean
EXCEPTIONS
       Overflow, List_Is_Null, Not_At_Head
  END
  OPERATOR Head_Of SPECIFICATION
    INPUT
The_List : List
    OUTPUT
Result : Item
EXCEPTIONS
       Overflow, List_Is_Null, Not_At_Head
  OPERATOR Tail_Of
SPECIFICATION
INPUT
The_List : List
    OUTPUT
Result : List
EXCEPTIONS
       Overflow, List_Is_Null, Not_At_Head
  END
  OPERATOR Predecessor_Of
  SPECIFICATION
INPUT
The_List : List
    OUTPUT
Result : List
EXCEPTIONS
       Overflow, List_Is_Null, Not_At_Head
  END
END
IMPLEMENTATION ADA List_Double_Unbounded_Unmanaged
```

LIST SINGLE BOUNDED MANAGED

```
generic
type Item is private;
The_Size : in Positive;
package List_Single_Bounded_Managed is

type List is private;

Null_List : constant List;

procedure Copy (From_The_List : in List;
To_The_List : in out List);
procedure Clear (The_List : in out List);
procedure Construct (The_Item : in Ltem;
And_The_List : in out List);
procedure Set_Head (Of_The_List : in out List);
procedure Swap_Tail (Of_The_List : in out List;
And_The_List : in out List;
And_The_List : in out List;
Procedure Swap_Tail (Of_The_List : in out List;
And_The_List : in out List;
Regult : in List;
Right : in List;
Result : out Boolean);
procedure Length_Of (The_List : in List;
```

LIST SINGLE BOUNDED MANAGED

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
                                                                       (3) (ii)
package body List_Single_Bounded_Managed is
      type Node is
                 The_Item : Item;
Next : List;
            end record:
     Heap : array(Positive range 1 .. The_Size) of Node;
     procedure Free (The_List : in out List) is
    Temporary_Node : List;
      begin
            n
while The_List /= Null_List loop
    Temporary_Node := The_List;
    The_List := Heap(The_List.The_Head).Next;
                 Heap(Temporary_Node.The_Head).Next := Free_List;
Free_List := Temporary_Node;
            end loop;
      end Free;
      function New Item return List is
            Temporary_Node : List;
     begin
if Free_List = Null_List then
if Storage Error;
                 raise Storage_Error;
     else
    Temporary_Node := Free_List;
    Free_List := Heap(Free_List.The_Head).Next;
    Heap(Temporary_Node.The_Head).Next := Null_List;
    return Temporary_Node;
end if;
end New_Item;
            else
     begin
           end loop;
end if;
     exception
when Storage_Error =>
raise Overflow;
     procedure Clear (The_List : in out List) is begin
            ree(The_List);
      end Clear;
     procedure Construct (The_Item : in Item;
And_The_List : in out List) is
Temporary_Node : List;
      begin
            Temporary_Node := New_Item;
           Heap(Temporary_Node: The_Head).The_Item := The_Item;
Heap(Temporary_Node: The_Head).Next := And_The_List;
And_The_List := Temporary_Node;
      exception
when Storage_Error =>
raise Overflow;
      end Construct:
      Heap(Of_The_List.The_Head).The_Item := To_The_Item;
      exception
```

```
when Constraint Error =>
     raise List_Is_Null;
end Set_Head;
     In
Temporary_Node := Heap(Of_The_List.The_Head).Next;
Heap(Of_The_List.The_Head).Next := And_The_List;
And_The_List := Temporary_Node;
     and_The_List := Temporar
exception
   when Constraint_Error =>
        raise List_Is_Null;
end Swap_Tail;
-- modified by Vincent Hong and Tuan Nguyen
-- date: 9 April 1995
     adding procedures to replace functions
                                        (Left : in List;
Right : in List;
Result : out Boolean) is
     procedure Is_Equal
     Result := Is_Equal (Left,Right);
end Is_Equal;
                                       (The_List : in List;
Result : out Natural) is
     procedure Length_Of
     begin
     Result := Length_Of
end Length_Of;
                                             (The List):
                                       (The_List : in List;
Result : out Boolean) is
     procedure Is_Null
     Result := Is_Null
end Is_Null;
                                             (The List):
                                       (The_List : in List;
Result : out Item) is
     procedure Head_Of
     begin
          Result := Head Of
                                             (The List):
     end Head_Of;
                                       (The_List : in List;
Result : out List) is
     procedure Tail_Of
     Result := Tail_Of
end Tail_Of;
                                             (The List);
-- end of modification
     in
while Left_Index /= Null_List loop
    if Heap(Left_Index.The_Head).The_Item /=
        Heap(Right_Index.The_Head).The_Item then
        return False;
    end if;
     begin
                enu 11;
Left_Index := Heap(Left_Index.The_Head).Next;
Right_Index := Heap(Right_Index.The_Head).Next;
     end loop;
return (Right_Index = Mull_List);
exception
when Constraint_Error =>
                return False;
     end Is_Equal;
     function Length_Of (The_List : in List) return Natural is
          Count : Natural := 0;
Index : List := The_List;
          while Index /= Null_List loop
          Count := Count + 1;
Index := Heap(Index.The_Head).Next;
end loop;
           return Count;
     end Length_Of;
     function Is_Null (The_List : in List) return Boolean is
     return (The_List = Null_List);
end Is_Null;
     function Head_Of (The_List : in List) return Item is
          return Heap(The_List.The_Head).The_Item;
     exception when Constraint_Error =>
     raise List_Is_Null;
end Head_Of;
      function Tail_Of (The_List : in List) return List is
```

return Heap(The_List.The_Head).Next;
exception
 when Constraint_Error =>
 raise List_Ts_Null;
end Tail_Of;

begin

LIST SINGLE BOUNDED MANAGED

PSDL

TYPE List_Single_Bounded_Managed PECIFICATION
GENERIC
Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
INPUT
From_The_List : List,
To_The_List : List
OUTPUT To_The_List : List
EXCEPTIONS
Overflow, List_Is_Null
END
OPERATOR Clear
SPECIFICATION
INPUT
The_List : List
OUTPUT
The_List : List
EXCEPTIONS Overflow, List_Is_Null
END CONTINUE TO THE CONTINUE T
END
OPERATOR Construct
SPECIFICATION
INPUT
The_Item : Item,
And_The_List : List
OUTPUT
And_The_List : List EXCEPTIONS
Overflow, List_Is_Null
END
212
OPERATOR Set_Head
SPECIFICATION
INPUT
Of_The_List : List, To_The_Item : Item
OUTPUT
Of_The_List : List
EXCEPTIONS
Overflow, List_Is_Null
END .
OPERATOR Swap_Tail
SPECIFICATION INPUT
Of_The_List : List,
And_The_List : List
OUTPUT
Of The List : List,
And_The_List : List
EXCEPTIONS
Overflow, List_Is_Null

```
OPERATOR Is_Equal
SPECIFICATION
INPUT
Left: List,
Right: List
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Length_Of
SPECIFICATION
INFUT
The_List: List
OUTPUT
Result: Natural
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Is_Null
SPECIFICATION
INFUT
The_List: List
OUTPUT
Result: Boolean
EXCEPTIONS
OVERFLOW, List_Is_Null
END

OPERATOR Head_Of
SPECIFICATION
INFUT
The_List: List
OUTPUT
Result: List
OUTPUT
Result: Item
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Tail_Of
SPECIFICATION
INPUT
The_List: List
OUTPUT
Result: List
OVERATOR Tail_Of
SPECIFICATION
INPUT
The_List: List
OUTPUT
Result: List
EXCEPTIONS
Overflow, List_Is_Null
END

END

IMPLEMENTATION ADA List_Single_Bounded_Managed
END
```

LIST SINGLE UNBOUNDED MANAGED

```
procedure Is_Null (The_List: in List;
Result : out Natural);
(The_List: in List;
Result : out Boolean);
(The_List: in List;
Result : out Item);
(The_List: in List;
Result : out Item);
(The_List: in List;
Result : out List);

-- end of modification

function Is_Equal (Left : in List;
Right : in List) return Boolean;
function Length_Of (The_List: in List) return Natural;
function Is_Null (The_List: in List) return Boolean;
function Head_Of (The_List: in List) return Boolean;
function Tail_Of (The_List: in List) return Item;
function Tail_Of (The_List: in List) return List;

Overflow : exception;
List_Is_Null: exception;

private
    type Node;
    type List is access Node;
    Null_List: constant List := null;
end List_Single_Unbounded_Managed;
```

LIST SINGLE UNBOUNDED MANAGED

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
- restrictions as set forth is absolutions (5)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body List_Single_Unbounded_Managed is
      type Node is
             record
             The Item : Item;
Next : List;
end record;
      procedure Free (The_Node : in out Node) is
begin
      null;
end Free;
      The_Node.Next := To_Next;
end Set_Next;
       function Next_Of (The_Node : in Node) return List is
             return The_Node.Next;
      end Next_Of;
      Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
     in Node_Manager.Free(To_The_List);

if From_The_List /= null then

To_The_List := Node_Manager.New_Item;

To_The_List.The_Item := From_Index.The_Item;

To_Index := To_The_List;

From_Index := From_Index.Next;

while From_Index := Node_Manager.New_Item;

To_Index.Next := Node_Manager.New_Item;

To_Index := To_Index.Next;

To_Index.The_Item := From_Index.The_Item;

From_Index := From_Index.Next;

end loop;

end if;

eption
       begin
       exception
             when Storage_Error => raise Overflow;
      end Copy;
         rocedure Clear (The_List : in out List) is
            Node_Manager.Free(The_List);
     begin
             Temporary_Node := Node_Manager.New_Item;
             Temporary_Node: "Node_Hanger Hew_I
Temporary_Node. The_Item: The_Item;
Temporary_Node. Next: And_The_List;
And_The_List: Temporary_Node;
      exception
when Storage_Error =>
raise Overflow;
end Construct;
      Of_The_List.The_Item := To_The_Item;
      exception
when Constraint_Error =>
raise List_Is_Null;
end Set_Head;
     procedure Swap_Tail (Of_The_List : in out List;
And_The_List : in out List) is
```

```
Temporary_Node : List;
     begin
           In
Temporary_Node := Of_The_List.Next;
Of_The_List.Next := And_The_List;
And_The_List := Temporary_Node;
     exception
when Constraint_Error =>
raise List_Is_Null;
end Swap_Tail;
-- modified by Vincent Hong and Tuan Nguyen
-- date: 9 April 1995
-- adding procedures to replace functions
                                                     : in List;
: in List;
     procedure Is Equal
                                          Right
                                         Result : out Boolean) is
     begin
           Result := Is_Equal (Left,Right);
     end Is_Equal;
                                        (The_List : in List;
Result : out Natural) is
     procedure Length_Of
                                             (The_List);
           Result := Length Of
     end Length_Of;
                                        (The_List : in List;
Result : out Boolean) is
     procedure Is_Null
     begin
     Result := Is_Null
end Is_Null;
                                              (The List):
                                        (The_List : in List;
Result : out Item) is
     procedure Head Of
                                             (The_List);
           Result := Head Of
     end Head_Of;
                                        (The_List : in List;
Result : out List) is
     procedure Tail Of
     begin
    Result := Tail_Of
end Tail_Of;
                                            (The_List);
-- end of modification
    begin
          in
while Left_Index /= null loop
   if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
end if;
                end 11;
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
          end loop;
return (Right_Index = null);
     exception
when Constraint_Error =>
return False;
     end Is Equal:
     function Length_Of (The_List : in List) return Natural is
   Count : Natural := 0;
   Index : List := The_List;
           while Index /= null loop
Count := Count + 1;
Index := Index.Next;
     end loop;
return Count;
end Length_Of;
     function Is_Null (The_List : in List) return Boolean is
     begin
          return (The_List = null);
     end Is_Null;
     function Head_Of (The_List : in List) return Item is
           return The_List.The_Item;
     exception
when Constraint_Error =>
raise List_Is_Null;
     function Tail_Of (The_List : in List) return List is
          return The_List.Next;
     exception when Constraint_Error => raise List_Is_Null; end Tail_Of;
```

end List_Single_Unbounded_Managed;

LIST SINGLE UNBOUNDED MANAGED

PSDL

TYPE List_Single_Unbounded_Managed SPECIFICATION GENERIC
Item : PRIVATE_TYPE
OPERATOR Copy SPECIFICATION
INPUT
From_The_List : List, To_The_List : List OUTPUT
To_The_List : List
EXCEPTIONS Overflow, List_Is_Null
END
OPERATOR Clear
SPECIFICATION
INPUT The_List : List
OUTPUT
The_List : List
EXCEPTIONS
Overflow, List_Is_Null
END
OPERATOR Construct SPECIFICATION INPUT
The_Item : Item,
And The List : List OUTPUT
And_The_List : List EXCEPTIONS
Overflow, List_Is_Null
OPERATOR Set_Head SPECIFICATION
INPUT Of The List : List,
To_The_Item : Item OUTPUT
Of_The_List : List
EXCEPTIONS Overflow, List_Is_Null
END COVERTION, LIST_IS_NUIT
OPERATOR Swap_Tail
SPECIFICATION
INPUT
Of_The_List : List, And_The_List : List
OUTPUT
Of_The_List : List,
And_The_List : List
EXCEPTIONS Overflow, List_Is_Null
Overilow, Disc_is_Null

```
OPERATOR Is_Equal
SPECIFICATION
INPUT
Left: List,
Right: List
OUTFUT
Result: Boolean
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Length_Of
SPECIFICATION
INPUT
The_List: List
OUTFUT
Result: Natural
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Is_Null
SPECIFICATION
INPUT
The_List: List
OUTFUT
Result: Boolean
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Head_Of
SPECIFICATION
INPUT
The_List: List
OUTFUT
Result: Item
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Tail_Of
SPECIFICATION
INPUT
The_List: List
OUTFUT
Result: List
OUTFUT
Result: Item
EXCEPTIONS
Overflow, List_Is_Null
END

OPERATOR Tail_Of
SPECIFICATION
INPUT
The_List: List
OUTFUT
Result: List
EXCEPTIONS
Overflow, List_Is_Null
END
```

LIST SINGLE UNBOUNDED UNMANAGED

LIST SINGLE UNBOUNDED UNMANAGED

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
                     "Restricted Rights Legend"
 -- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
 package body List_Single_Unbounded_Unmanaged is
       type Node is
             record
The_Item : Item;
Next : List;
end record;
      begin
    if From_The_List = null then
        To_The_List := null;
                   To_Index := To_The_List;
From_Index := To_The_List;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
             Nex ro_Index := To_Index.Next;
    From_Index := From_Index.Next;
end loop;
end if;
pption
                                                                                  => null);
       exception
when Storage_Error =>
raise Overflow;
       end Copy;
       procedure Clear (The_List : in out List) is
begin
             The_List := null;
      procedure Construct (The_Item : in Item; And_The_List : in out List) is
             exception
    when Storage_Error =>
        raise Overflow;
end Construct;
       Of_The_List.The_Item := To_The_Item;
       exception
when Constraint_Error =>
raise List_Is_Null;
       end Set Head:
       In
Temporary_Node := Of_The_List.Next;
Of_The_List.Next := And_The_List;
And_The_List := Temporary_Node;
       And_The_List := Temporar;
exception when Constraint_Error =>
raise List_Is_Null;
end Swap_Tail;
      modified by Vincent Hong and Tuan Nguyen
date: 9 April 1995
adding procedures to replace functions
```

```
Left : in List;
Right : in List;
Result : out Boolean) is
    procedure Is_Equal
                                    (Left
    Result := Is_Equal end Is_Equal;
                                    (Left, Right);
    procedure Length Of
                                   (The_List : in List;
Result : out Natural) is
    begin
         Result := Length_Of
                                        (The List):
    end Length_Of;
                                   (The_List : in List;
Result : out Boolean) is
    procedure Is Null
    Result := Is_Null
end Is Null;
                                        (The_List);
                                   (The_List : in List;
Result : out Item) is
    Result := Head_Of
end Head_Of;
                                        (The_List);
                                   (The_List : in List;
Result : out List) is
    procedure Tail_Of
    begin
    Result := Tail_Of
end Tail_Of;
                                        (The_List);
-- end of modification
    un
while Left_Index /= null loop
if Left_Index.The_Item /= Right_Index.The_Item then
              return False;
end if;
Left_Index := Left_Index.Next;
              Right_Index := Right_Index.Next;
         end loop;
return (Right_Index = null);
    exception
         when Constraint_Error =>
    return False;
end Is_Equal;
    while Index /= null loop
Count := Count + 1;
Index := Index.Next;
         end loop;
    return Cou
end Length_Of;
    function Is_Null (The_List : in List) return Boolean is
    return (The_List = null);
end Is_Null;
     function Head Of (The List : in List) return Item is
    begin
         return The_List.The_Item;
    exception
when Constraint_Error =>
raise List_Is_Null;
end Head_Of;
     function Tail_Of (The_List : in List) return List is
    begin
    return The_List.Next;
    exception
when Constraint_Error =>
raise List_Is_Null;
end Tail_Of;
```

end List_Single_Unbounded_Unmanaged;

LIST SINGLE UNBOUNDED UNMANAGED

PSDL

```
TYPE List_Single_Unbounded_Unmanaged SPECIFICATION
   GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
        INPUT
From_The_List : List,
To_The_List : List
OUTPUT
        OUTPUT
To_The_List : List
EXCEPTIONS
Overflow, List_Is_Null
   OPERATOR Clear
SPECIFICATION
INPUT
The_List : List
        OUTPUT
The_List : List
        EXCEPTIONS
           Overflow, List_Is_Null
   OPERATOR Construct
SPECIFICATION
       INPUT
The_Item : Item,
And_The_List : List
       OUTPUT
And_The_List : List
EXCEPTIONS
Overflow, List_Is_Null
   END
   OPERATOR Set_Head
SPECIFICATION
INPUT
Of_The_List : List,
To_The_Item : Item
OUTPUT
       OUTPUT
Of_The_List : List
EXCEPTIONS
Overflow, List_Is_Null
    END
    OPERATOR Is_Equal
SPECIFICATION
INPUT
           Left : List,
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED ITERATOR

```
(The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges);
        procedure Range Of
       end of modication
        function Is_Equal (Left : in Map; Right : in Map) return Boolean; function Extent_Of (The_Map : in Map) return Natural; function Is_Bound (The_Map : in Map) return Boolean; function Is_Bound (The_Domain : in Domain; In_The_Map : in Map) return Boolean; function Range_Of (The_Domain : in Domain; In_The_Map : in Map) return Boolean;
                                                 In_The_Map : in Map) return Ranges;
        generic
                with procedure Process (The_Domain : in Domain;
The_Range : in Ranges;
Continue : out Boolean);
        procedure Iterate (Over_The_Map : in Map);
        Overflow
         Domain Is Not Bound : exception;
        Multiple_Binding
private
        type State is (Empty, Deleted, Bound); type Node is
                record
                The_State : State := Empty;
The_Domain : Domain;
The_Range : Ranges;
end record;
        end record;
type Items is array (Positive range <>) of Node;
type Map(The_Size : Positive) is
    record
    The_Items : Items(1 .. The_Size);
    The_Count : Natural := 0;
                end record:
end Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
                     "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FRR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator
      In_The_Map.The_Size;
            Temporary_Index : Positive;
Temporary_Bucket : Natural;
      begin
             Temporary_Bucket := 0;
for Index in In_The_Map.The_Items'Range loop
Temporary_Index := ((Index + Initial_Probe - 2) mod In_The_Map.The_Size) +
1:
                   case In_The_Map.The_Items(Temporary_Index).The_State is
                          when Empty =>
  if Temporary_Bucket = 0 then
   Temporary_Bucket := Temporary_Index;
  end if;
                                The_Bucket := Temporary_Bucket; return;
                          when Deleted =>
                                if Temporary_Bucket = 0 then
    Temporary_Bucket := Temporary_Index;
end if;
                         when Bound
return:
                                end if
            end case;
end loop;
The_Bucket := Temporary_Bucket;
      procedure Copy (From_The_Map : in Map;
To_The_Map : in out Map) is
            To_The_Map
The_Bucket : Natural;
      begin
            if From_The_Map.The_Count > To_The_Map.The_Size then
                   raise Overflow;
                   for Index in To_The_Map.The_Items'Range loop
To_The_Map.The_Items(Index).The_State := Empty;
                   Find(From_The_Map.The_Items(Index).The_Domain,
    To_The_Map. The_Bucket);
To_The_Map.The_Items(The_Bucket) :=
    From_The_Map.The_Items(Index);
     From_The_Map.The_Items(Index):

end if;
end loop;
To_The_Map.The_Count := From_The_Map.The_Count;
end if;
end Copy;
      procedure Clear (The_Map : in out Map) is
            for Index in The_Map.The_Items'Range loop
The_Map.The_Items(Index).The_State := Empty;
             end loop;
The_Map.The_Count := 0;
      end Clear:
      procedure Bind (The_Domain : in Domain;
And_The_Range : in Ranges;
In_The_Map : in out Map) is
             The_Bucket : Natural;
            in
Find(The_Domain, In_The_Map, The_Bucket);
if In_The_Map.The_Items(The_Bucket).The_State = Bound then
    raise Multiple_Binding;
                 se
In_The_Map.The_Items(The_Bucket) :=
Node'(Bound, The_Domain, And_The_Range);
In_The_Map.The_Count := In_The_Map.The_Count + 1;
```

```
exception
          when Constraint_Error =>
    raise Overflow;
end Bind;
    procedure Unbind (The_Domain : in Domain;
In_The_Map : in out Map) is
The_Bucket : Natural;
     begin
         in
Find(The_Domain, In_The_Map, The_Bucket);
if In_The_Map. The_Items(The_Bucket).The_State = Bound then
In_The_Map. The_Items(The_Bucket).The_State := Deleted;
In_The_Map. The_Count := In_The_Map.The_Count - 1;
         raise Domain_Is_Not_Bound;
end if;
     exception
when Constraint_Error =>
raise Domain_Is_Not_Bound;
     end Unbind:
-- modified by Tuan Nguyen and Vincent Hong
    date: 8 April 1995
adding procedures to replace functions
                             (Left : in Map;
Right : in Map;
Result : out Boolean) is
     procedure Is_Equal (Left
         Result := Is_Equal(Left,Right);
     end Is_Equal;
    Result := Extent_Of(The_Map);
end Extent_Of;
    begin
    Result := Is_Empty(The_Map);
end Is_Empty;
    procedure Is_Bound (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Boolean) is
    begin
         Result := Is_Bound(The_Domain,In_The_Map);
     end Is_Bound;
    procedure Range_Of (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges) is
    Result := Range_Of(The_Domain,In_The_Map);
end Range_Of;
 - end of modification
    begin
         if Left.The_Count /= Right.The_Count then
              return False;
              Bound)
and then
       exit;
end if;
end loop;
return True;
end if;
eption
                                 (Left.The_Items(Index).The_Domain =
then
     exception
when Constraint_Error =>
              return False;
     end Is Equal:
     function Extent_Of (The_Map : in Map) return Natural is
     begin
         return The_Map.The_Count;
     end Extent_Of;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED ITERATOR

```
TYPE Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator SPECIFICATION
   GENERIC
      Domain : PRIVATE TYPE,
      Domain : PRIVATE_TYPE,
Ranges : PRIVATE_TYPE,
Hash_Of : FUNCTION[The_Domain : Domain, RETURN : Positive],
Hash_Of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positive]]
OPERATOR Copy
  OPERATOR Copy
SPECIFICATION
INPUT
From_The_Map : Map,
To_The_Map : Map,
      OUTPUT
To_The_Map : Map
EXCEPTIONS
         Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Clear
   SPECIFICATION
INPUT
The_Map : Map
      OUTPUT
The_Map : Map
EXCEPTIONS
         Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Bind
SPECIFICATION
INPUT
         The_Domain : Domain,
     The_Domain : Domain,
And_The_Range : Ranges,
In_The_Map : Map
OUTPUT
In_The_Map : Map
EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Unbind
SPECIFICATION
      INPUT
The_Domain : Domain,
In_The_Map : Map
OUTPUT
         In_The_Map : Map
      EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Is_Equal SPECIFICATION
      INPUT
Left : Map,
Right : Map
      OUTPUT
         Result : Boolean
      EXCEPTIONS
```

```
Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Extent_O.
SPECIFICATION
INPUT
The_Map : Map
OUTPUT
   OPERATOR Extent Of
        Result : Natural
     EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
   END
   OPERATOR IS_Empty
SPECIFICATION
     PECIFICATION
INPUT
The_Map : Map
OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Is_Bound SPECIFICATION
      INPUT
        NPUT
The_Domain : Domain,
In_The_Map : Map
     OUTPUT
     Result : Boolean
EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Range_Of SPECIFICATION
     INPUT
The_Domain : Domain,
In_The_Map : Map
     OUTPUT
     Result : Ranges
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Iterate
   SPECIFICATION
GENERIC
Process: PROCEDURE[The_Domain: in[t: Domain], The_Range: in[t: Ranges], Continue: out[t: Boolean]]
     INPUT
Over_The_Map : Map
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
INFIDENCE/NEATTON ADA
Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator
END
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED NONITERATOR

```
generic
type Domain is private;
type Ranges is private;
Number_Of_Buckets : in Positive;
with function Hash_Of (The_Domain : in Domain) return Positive;

-- modified by Tuan Nguyen and Vincent Hong
-- date: 8 April 1995
-- adding procedures to replace functions
with procedure Hash_Of (The_Domain : in Domain;
Result : out Positive);
-- end of modication

package Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator
is

type Map is limited private;

procedure Copy (Prom_The_Map : in out Map);
procedure Clear (The_Map : in out Map);
procedure Bind (The_Domain : in Domain;
And_The_Range : in Ranges;
In_The_Map : in out Map);
procedure Unbind (The_Domain : in Domain;
In_The_Map : in out Map);
-- modified by Tuan Nguyen and Vincent Hong
-- date: 8 April 1995
-- adding procedures to replace functions
procedure Is_Equal (Left : in Map;
```

```
Right : in Map;
Result : out Boolean);
procedure Extent_Of (The_Map : in Map;
Result : out Natural);
                                              procedure Is_Empty
                                                                                                   : out Boolean);
       procedure Is_Bound
                                             In_The_Map : in Map;
Result : out Boolean);
(The_Domain : in Domain;
In_The_Map : in Map;
       procedure Range_Of
                                                 Result
                                                                      : out Ranges):
-- end of modication
       function Is_Equal (Left
                                                                     : in Map:
       function Is_Equal (Left : in Map;
Right : in Map) return Boolean;
function Extent_Of (The_Map : in Map) return Natural;
function Is_Bumpty (The_Map : in Map) return Boolean;
function Is_Bound (The_Domain : in Domain;
In_The_Map : in Map) return Boolean;
function Range_Of (The_Domain : in Domain;
In_The_Map : in Map) return Ranges;
       Overflow
                                               : exception;
       Domain_Is_Not_Bound : exception;
Multiple_Binding : exception;
private
       rate
type Node;
type Structure is access Node;
type Structure is access Node;
type Map is array (Positive range 1 .. Number_Of_Buckets) of
end Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED NONITERATOR

```
--

-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch

-- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body
Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator is
          type Node is
                  Node 15
record
The_Domain : Domain;
The_Range : Ranges;
Nort : Structure;
                   end record;
         procedure Free (The_Node : in out Node) is
begin
                   null:
         procedure Set_Next (The_Node : in out Node;
To_Next : in Struc
                                                                                                   Structure) is
         begin
                   The Node.Next := To_Next;
          end Set_Next;
          function Next_Of (The_Node : in Node) return Structure is
          return The_Node.Next;
end Next_Of;
          package Node_Manager is new Storage_Manager_Sequential
                                                                                      rage_Manager_Sequential
(Item => Node,
Pointer => Structure,
Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
             rocedure Find (The_Domain : in Domain;
In_The_Map : in Map;
The_Bucket : out Positive;
Previous_Node : in out Structure;
Current_Node : in out Structure) is
Temporary_Bucket : Positive :=
(Hash_Of (The_Domain) mod
t Of Buckets) + 1:
          procedure Find (The_Domain
In_The_Map
The_Bucket
Number_Of_Buckets) + 1;
         ir Current_Node.The_Domain = The_Domain
    return;
else
    Previous_Node := Current_Node;
    Current_Node := Current_Node.Next;
end if;
end loop;
Find;
          end Find:
         begin
                   in
for Index in To_The_Map'Range loop
    Node_Manager.Free(To_The_Map(Index));
end loop;
for Index in From_The_Map'Range_loop
                            Index in From_The_Map'Range loop
From_Index := From_The_Map(Index);
if From_The_Map(Index) /= null then
To_The_Map(Index) := Node_Manager.New_Item;
To_The_Map(Index) := Node_Manager.New_Item;
To_The_Map(Index).The_Domain := From_Index.The_Domain;
To_The_Map(Index).The_Range := From_Index.The_Range;
To_Index := To_The_Map(Index);
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := Node_Manager.New_Item;
To_Index.Next := Node_Manager.New_Item;
To_Index.Next.The_Domain := From_Index.The_Domain;
To_Index.Next.The_Range := From_Index.The_Range;
To_Index := To_Index.Next;
From_Index := From_Index.Next;
end loop;
                   end loop;
end if;
end loop;
           exception
                   when Storage_Error => raise Overflow;
          end Copy:
```

```
procedure Clear (The_Map : in out Map) is begin
            for Index in The_Map'Range loop
   Node_Manager.Free(The_Map(Index));
            end loop;
          And_The_Range : in Domain;
And_The_Range : in cut Map) is
The_Bucket : Positive;
Previous_Node : Structure;
Temporary_Node : Structure;
In Find(The_Domain -
      procedure Bind (The_Domain
      begin
            Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
if Current_Node /= null then
raise Multiple_Binding;
           raise multiple_services

Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Domain := The_Domain;
Temporary_Node.The_Range := And_The_Range;
Temporary_Node.Next := In_The_Map(The_Bucket);
In_The_Map(The_Bucket) := Temporary_Node;
      exception
     when Storage_Error =>
raise Overflow;
end Bind;
     Previous_Node : Structure;
Current_Node : Structure;
      begin
            Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
            .Node);
if Previous_Node = null then
In_The_Map (The_Bucket) := Current_Node.Next;
            Previous_Node.Next := Current_Node.Next;
end if;
            Current_Node.Next := null;
Node_Manager.Free(Current_Node);
      exception
when Constraint_Error =>
                  raise Domain_Is_Not_Bound;
     modified by Tuan Nguyen and Vincent Hong
date: 8 April 1995
adding procedures to replace functions
      procedure Is_Equal (Left : in Map;
                                    Right : in Map;
Result : out Boolean) is
      begin
            Result := Is_Equal(Left,Right);
      end Is_Equal;
      Result := Extent_Of(The_Map);
end Extent_Of;
      Result := Is_Empty(The_Map);
end Is_Empty;
      procedure Is_Bound (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Boolean) is
            Result := Is_Bound(The_Domain,In_The_Map);
      procedure Range_Of (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges) is
      begin
            Result := Range_Of(The_Domain,In_The_Map);
      end Range_Of;
    end of modification
      function Is_Equal (Left : in Map;
Right : in Map) return Boolean is
Left_Index : Structure;
Right_Index : Structure;
Left_Count : Natural;
Right_Count : Natural;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED NONITERATOR

```
TYPE Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator SPECIFICATION GENERIC
     Domain : PRIVATE_TYPE,
     Hash_Of: PROVEDURE[The_Domain: Domain, RETURN: Positive],
Hash_Of: PROCEDURE[The_Domain: in[t: Domain], Result: out[t:
Positive]]
OPERATOR Copy
SPECIFICATION
     PECIFICATION
INPUT
From_The_Map : Map,
To_The_Map : Map
OUTFUT
To_The_Map : Map
EXCEPTIONS
         Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Clear SPECIFICATION
     INPUT
The_Map : Map
OUTPUT
The_Map : Map
      EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   END
   OPERATOR Bind
SPECIFICATION
     INPUT
The_Domain : Domain,
And_The_Range : Ranges,
In_The_Map : Map
      OUTPUT
     In_The_Map : Map
EXCEPTIONS
         Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Unbind SPECIFICATION
     INPUT
The_Domain : Domain,
In_The_Map : Map
     In_The_Map : Map
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   END
  OPERATOR Is_Equal
   SPECIFICATION
INPUT
```

```
Left : Map,
Right : Map
OUTPUT
Result : Boolean
      EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Extent_Of SPECIFICATION
     INPUT
The_Map : Map
     OUTPUT
        Result : Natural
     EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  EMD
  OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Map : Map
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Is_Bound SPECIFICATION
    The_Domain : Domain,
In_The_Map : Map
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Range_Of
  OPERATOR RANGE_OF
SPECIFICATION
INPUT
The_Domain : Domain,
In_The_Map : Map
OUTPUT
Result : Ranges
     EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Unbounded_Managed_NoniteratorEND
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
generic
      type Domain is private;
      type Ranges is private;
Number_Of_Buckets : in Positive;
with function Hash_Of (The_Domain : in Domain) return Positive;
      modified by Tuan Nguyen and Vincent Hong
      date: 8 April 1995
adding procedures to replace functions
      with procedure Hash_Of (The_Domain : in Domain;
Result : out Positive);
                                               Result
-- end of modication
package Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator is
       type Map is limited private;
      procedure Copy (From_The_Map : in out Map;
To_The_Map : in out Map);
procedure Clear
procedure Bind (The_Map : in out Map);
(The_Domain : in Domain;
                                   (The_Domain : in Domain And_The_Range : in Range In_The_Map : in out Map);
                                                                            Ranges;
      procedure Unbind (The_Domain In_The_Map
                                                             : in
                                                                            Domain:
                                                             : in out Map);
      modified by Tuan Nguyen and Vincent Hong
date: 8 April 1995
adding procedures to replace functions
      procedure Is_Equal (Left : in Map;
Right : in Map;
Result : out Boolean);
procedure Extent_Of (The_Map : in Map;
Result : out Natural);
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                         "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator is
        type Node is
               record
                        The_Domain : Domain;
The_Range : Ranges;
Next : Structure;
                end record;
        procedure Free (The_Node : in out Node) is begin
               null;
        end Free;
       procedure Set_Next (The_Node : in out Node;
To_Next : in Struct
                                                                                     Structure) is
                The_Node.Next := To_Next;
        end Set Next:
        function Next_Of (The_Node : in Node) return Structure is
        begin
                return The_Node.Next;
        end Next_Of;
       Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
        procedure Find (The_Domain
In_The_Map
The_Bucket
                                                                    : in
                                                                                       Domain;
           Number_Of_Buckets) + 1;
        begin
               in
The_Bucket := Temporary_Bucket;
Current_Node := In_The_Map(Temporary_Bucket);
while Current_Node /= null loop
    if Current_Node.The_Domain = The_Domain then
               else Previous_Node := Current_Node;
Current_Node := Current_Node.Next;
end if;
end loop;
Find;
        end Find;
       begin
               for Index in To_The_Map'Range loop
   Node_Manager.Free(To_The_Map(Index));
end loop;
               end loop;
for Index in From_The_Map'Range loop
    From_Index := From_The_Map(Index);
    if From_The_Map(Index) /= null then
        To_The_Map(Index) /= null then
        To_The_Map(Index) := Node_Manager.New_Item;
        To_The_Map(Index).The_Domain := From_Index.The_Domain;
        To_The_Map(Index).The_Range := From_Index.The_Range;
        To_Index := To_The_Map(Index);
        From_Index := From_Index.Next;
        while From_Index /= null loop
        To_Index.Next := Node_Manager.New_Item;
        To_Index.Next.The_Domain := From_Index.The_Domain;
        To_Index.Next.The_Range := From_Index.The_Range;
        To_Index.The_Range := From_Index.The_Range;
        To_Index := To_Index.Next;
        end loop;
end loop;
                        end loop;
end if;
                end loop;
        exception when Storage_Error =>
                        raise Overflow;
        end Copy:
```

```
procedure Clear (The_Map : in out Map) is begin
             for Index in The_Map'Range loop
Node_Manager.Free(The_Map(Index));
end loop;
      end Clear;
     procedure Bind (The_Domain : in Domain;
And_The_Range : in Ranges;
In_The_Map : in out Map) is
The_Bucket : Positive;
Previous_Node : Structure;
Current_Node : Structure;
Temporary_Node : Structure;
      begin
             Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current Node):
             if Current_Node /= null then
raise Multiple_Binding;
                   Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Domain := The_Domain;
Temporary_Node.The_Range := And_The_Range;
Temporary_Node.Next := In_The_Map(The_Bucket);
In_The_Map(The_Bucket) := Temporary_Node;
             end if;
      exception
when Storage_Error =>
raise Overflow;
end Bind;
      procedure Unbind (The_Domain : in __Domain;
	In_The_Map : in out Map) is
	The_Bucket : Positive;
	Previous_Node : Structure;
	Current_Node : Structure;
      begin
             Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Previous_Node.Next := Current_Node.Next;
             end if;
current_Node.Next := null;
Node_Manager.Free(Current_Node);
      exception
when Constraint_Error =>
raise Domain_Is_Not_Bound;
end Unbind;
     modified by Tuan Nguyen and Vincent Hong date: 8 April 1995
      adding procedures to replace functions
                                       (Left : in Map;
Right : in Map;
Result : out Boolean) is
      procedure Is_Equal (Left
             Result := Is_Equal(Left,Right);
      end Is_Equal;
      Result := Extent_Of(The_Map);
      end Extent_Of;
      procedure Is_Empty (The_Map : in Map;
Result : out Boolean) is
      begin
      Result := Is_Empty(The_Map);
end Is_Empty;
      procedure Is_Bound (The_Domain : in Domain;
	In_The_Map : in Map;
	Result : out Boolean) is
      Result := Is_Bound(The_Domain,In_The_Map);
end Is_Bound;
      procedure Range_Of (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges) is
      Result := Range_Of(The_Domain,In_The_Map);
end Range_Of;
-- end of modification
      function Is_Equal (Left : in Map;
Right : in Map) return Boolean is
Left_Index : Structure;
Right_Index : Structure;
Left_Count : Natural;
Right_Count : Natural;
begin
```

```
for Index in Left'Range loop
   if (Left(Index) = null) xor (Right(Index) = null) then
        return False;
                           exit:
                                           else
                                          Right_Index := Right_Index.Next;
end if;
                                   end loop;
if Left_Index.The_Range /= Right_Index.The_Range
then
                                          return False;
                                   else
Left_Index := Left_Index.Next;
Left_Count := Left_Count + 1;
                                   end if;
                           end if;
end loop;
Right_Index := Right(Index);
Right_Count := 0;
while Right_Index /= null loop
Right_Index := Right_Index.Next;
Right_Count := Right_Count + 1;
                            end loop;
if Left_Count /= Right_Count then
return False;
end if;
             end if;
end loop;
return True;
      exception
when Constraint_Error =>
return False;
       end Is Equal:
       function Extent_Of (The_Map : in Map) return Natural is
   Count : Natural := 0;
   Temporary_Node : Structure;
       begin
             for Index in The_Map'Range loop
   Temporary_Node := The_Map(Index);
   while Temporary_Node /= null loop
    Count := Count + 1;
        Temporary_Node := Temporary_Node.Next;
   and loop
       end loop;
end loop;
return Count;
end Extent_Of;
       function Is_Empty (The_Map : in Map) return Boolean is
       begin
return (The_Map = Map'(others => null));
```

```
end Is Empty:
      function Is_Bound (The_Domain : in Domain;
In_The_Map : in Map) return Boolean is
The_Bucket : Positive;
Previous_Node : Structure;
Current_Node : Structure;
       begin
              Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
return (Current_Node /= null);
       end Is_Bound;
      The_Bucket : Positive;
Previous_Node : Structure;
Current_Node : Structure;
       begin
Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node):
              return Current Node. The Range;
      return Current_Node.inc_Compg.,
exception
when Constraint_Error =>
raise Domain_Is_Not_Bound;
end Range_Of;
      procedure Iterate (Over_The_Map : in Map) is
   The_Bucket : Positive := Over_The_Map'Last;
   The_Node : Structure;
   Continue : Boolean;
       begin
              in
for The_Iterator in Over_The_Map'Range loop
    if Over_The_Map(The_Iterator) /= null then
        The_Bucket := The_Iterator);
        The_Node := Over_The_Map(The_Iterator);
             exit;
end if;
end loop;
while The_Node /= null loop
Process(The_Node.The_Domain, The_Node.The_Range,
                     exit when not Continue;
                     The Node := The Node Next;

if The Node = null then
for The_Iterator in (The_Bucket + 1) ...
Over_The_Map'Last loop
if Over_The_Map(The_Iterator) /= null then
                                  The_Bucket := The_Iterator;
The_Node := Over_The_Map(The_Iterator);
exit;
end if;
       end i
end loop;
end loop;
end loop;
end lterate;

Vap_Sim-
 end Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
TYPE Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
GENERIC
      Domain : PRIVATE_TYPE,
      Domain : FRIVATE_TYPE,
Ranges : FRIVATE_TYPE,
Hash_of : FUNCTION[The_Domain : Domain, RETURN : Positive],
Hash_of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positivell
  OPERATOR COPY
SPECIFICATION
     From_The_Map : Map,
To_The_Map : Map
OUTPUT
     To_The_Map : Map
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Clear
  SPECIFICATION
INPUT
     The_Map : Map
        The_Map : Map
     EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Bind
SPECIFICATION
     The Domain : Domain,
And The Range : Ranges,
In The Map : Map
OUTPUT
     OUTPUT
In_The_Map: Map
EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Unbind
SPECIFICATION
     INPUT
     The_Domain : Domain,
In_The_Map : Map
OUTPUT
        In_The_Map : Map
     EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR IS_Equal SPECIFICATION
     INPUT
Left : Map,
Right : Map
     OUTPUT
     Result : Boolean
EXCEPTIONS
```

```
Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Extent_Of
   SPECIFICATION
INPUT
     The_Map : Map
OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR IS_Empty SPECIFICATION
     INPUT
        The_Map : Map
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Is_Bound
   SPECIFICATION
     PECIFICATION
INPUT
The_Domain : Domain,
In_The_Map : Map
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Range Of
   SPECIFICATION
INPUT
The_Domain : Domain,
In_The_Map : Map
     OUTPUT
Result : Ranges
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Iterate
   SPECIFICATION
     GENERIC
Process: PROCEDURE[The_Domain : in[t : Domain], The_Range : in[t : Ranges], Continue : out[t : Boolean]]
     INPUT
     Over_The_Map : Map
EXCEPTIONS
       Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
INFLEMENTATION ADA Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator END
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
generic
type Domain is private;
type Ranges is private;
Number_Of_Buckets : in Positive;
with function Hash_Of (The_Domain : in Domain) return Positive;

-- modified by Tuan Nguyen and Vincent Hong
-- date: 8 April 1995
-- adding procedures to replace functions
with procedure Hash_Of (The_Domain : in Domain;
Result : out Positive);
-- end of modication

package
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator is
type Map is limited private;
procedure Copy (From_The_Map : in Map;
To_The_Map : in out Map);
procedure Clear (The_Map : in out Map);
procedure Bind (The_Domain : in Domain;
And_The_Range : in Ranges;
In_The_Map : in out Map);
procedure Unbind (The_Domain : in Domain;
In_The_Map : in out Map);
-- modified by Tuan Nguyen and Vincent Hong
-- date: 8 April 1995
-- adding procedures to replace functions
procedure Is_Equal (Left : in Map;
```

```
Right : in Map;
Result : out Boolean);

procedure Extent_Of (The_Map : in Map;
Result : out Natural);

procedure Is_Empty (The_Map : in Map;
Result : out Boolean);

procedure Is_Bound (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Boolean);

(The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges);

-- end of modication

function Is_Equal (Left : in Map;
Right : in Map) return Boolean;

function Is_Empty (The_Map : in Map) return Boolean;

function Is_Bound (The_Domain : in Domain;
In_The_Map : in Map) return Boolean;

function Range_Of (The_Domain : in Domain;
In_The_Map : in Map) return Boolean;

function Range_Of (The_Domain : in Domain;
In_The_Map : in Map) return Ranges;

Overflow : exception;
Domain_Is_Not_Bound : exception;
Multiple_Binding : exception;

private
type Mode;
type Mode;
type Map is array (Positive range 1 . Number_Of_Buckets) of

Structure;
end Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
-- (C) Copyright 1985, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
                        "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator is
       type Node is
               record
                      The Domain : Domain:
                      The_Range : Ranges;
Next : Structure;
               end record;
       procedure Find (The_Domain : in In_The_Map : in
                                                                                 Domain;
                                      In_The_Map
The_Bucket
                                                                                 Map;
Positive;
          In_The_Map : In map;
The_Bucket : out Positive;
Previous_Node : in out Structure;
Current_Node : in out Structure) is
Temporary_Bucket : Positive :=
(Hash_Of (The_Domain) mod
Number_Of_Buckets) + 1;
       begin
              in
The_Bucket := Temporary_Bucket;
Current_Node := In_The_Map(Temporary_Bucket);
while Current_Node /= null loop
    if Current_Node.The_Domain = The_Domain then
                             return:
                      Previous_Node := Current_Node;
Current_Node := Current_Node.Next;
end if;
               end loop;
       end Find;
       procedure Copy (From_The_Map : in Map;
To_The_Map : in out Map) is
              To_Index : Structure;
To_Index : Structure;
              for Index in From_The_Map'Range loop
    From_Index := From_The_Map(Index);
    if From_The_Map(Index) = null then
        To_The_Map(Index) := null;
                      else
To_The_Map(Index) := new Node'
(The_Domain =>
From Index. The Domain,
                                                                           The Range =>
From_Index.The_Range,
                                                                                                => null);
                              To_Index := To_The_Map(Index);
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := new_Node'
                                                                          (The Domain =>
From_Index.The_Domain,
From_Index.The_Range,
                                                                                                => null);
                                                                           Next
                                     To_Index := To_Index.Next;
From_Index := From_Index.Next;
             end loop;
end loop;
end loop;
eption
       exception
when Storage_Error =>
raise Overflow;
       end Copy;
       procedure Clear (The_Map : in out Map) is begin
               The_Map := Map'(others => null);
       procedure Bind (The_Domain : in Domain;
And_The_Range : in Ranges;
In_The_Map : in out Map) is
The_Bucket : Positive;
Previous_Node : Structure;
Current_Node : Structure;
       begin
Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current Node):
               Node);
if Current_Node /= null then
    raise Multiple_Binding;
               else
                      In_The_Map(The_Bucket) := new Node'
```

```
(The_Domain => The_Domain,
The_Range => And_The_Range,
Next =>
In_The_Map(The_Bucket));
     end if;
exception
when Storage_Error =>
               raise Overflow;
     end Bind;
    procedure Unbind (The_Domain : in Domain;
In_The_Map : in out Map) is
The_Bucket : Fositive;
Previous_Node : Structure;
Current_Node : Structure;
    begin
Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Find(inc_structure)
Current_Node);
    if Previous_Node = null then
        In_The_Map (The_Bucket) := Current_Node.Next;
          Previous_Node.Next := Current_Node.Next;
end if;
     exception
when Constraint_Error =>
               raise Domain_Is_Not_Bound;
     end Unbind;
-- modified by Tuan Nguyen and Vincent Hong
     date: 8 April 1995
adding procedures to replace functions
     procedure Is_Equal (Left : in Map;
Right : in Map;
Result : out Boolean) is
          Result := Is_Equal(Left,Right);
     end Is Equal;
     begin
     Result := Extent_Of(The_Map);
end Extent_Of;
     Result := Is_Empty(The_Map);
     end Is_Empty;
     Result := Is_Bound(The_Domain,In_The_Map); end Is_Bound;
     procedure Range_Of (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges) is
     Result := Range_Of(The_Domain, In_The_Map);
end Range_Of;
-- end of modification
     function Is_Equal (Left : in Map;
Right : in Map) return Boolean is
         Right: in
Left_Index: Structure;
Right_Index: Structure;
Left_Count: Natural;
Right_Count: Natural;
          for Index in Left'Range loop
   if (Left(Index) = null) xor (Right(Index) = null) then
     return False;
   else
     begin
                   Right_Index.The_Domain) then
                                  exit;
                              else
                                  Right_Index := Right_Index.Next;
                         end if;
end loop;
if Left_Index.The_Range /= Right_Index.The_Range
                              return False;
                         else
                              Left_Count := Left_Count + 1;
                    end if;
end loop;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
{\tt TYPE\ Map\_Simple\_Noncached\_Sequential\_Unbounded\_Unmanaged\_Noniterator} \\ {\tt SPECIFICATION}
   GENERIC
      EMERAL
Domain: PRIVATE_TYPE,
RANGES: PRIVATE_TYPE,
Hash_Of: FUNCTION[The_Domain: Domain, RETURN: Positive],
Hash_Of: PROCEDURE[The_Domain: in[t: Domain], Result: out[t:
Positive]]
OPERATOR Copy
SPECIFICATION
      PECIFICATION
INPUT
From_The_Map : Map,
To_The_Map : Map
OUTFUT
To_The_Map : Map
EXCEPTIONS
          Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Clear
SPECIFICATION
      INPUT
         The_Map : Map
      OUTFUT
The_Map : Map
EXCEPTIONS
          Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Bind SPECIFICATION
      INPUT
      INPUT
The_Domain : Domain,
And_The_Range : Ranges,
In_The_Map : Map
OUTFUT
         In_The_Map : Map
      EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
   END
   OPERATOR Unbind
SPECIFICATION
      INPUT
The_Domain : Domain,
In_The_Map : Map
OUTPUT
      In_The_Map : Map EXCEPTIONS
          Overflow, Domain_Is_Not_Bound, Multiple_Binding
   END
   OPERATOR Is_Equal SPECIFICATION
```

```
Left : Map.
     Right : Map
OUTPUT
Result : Boolean
     EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Extent_Of SPECIFICATION
     INPUT
The_Map : Map
OUTPUT
       Result : Natural
     EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Is_Empty
SPECIFICATION
     INPUT
The_Map : Map
OUTPUT
     Result : Boolean
EXCEPTIONS
       Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Is_Bound SPECIFICATION
     INPUT
The_Domain : Domain,
In_The_Map : Map
     OUTPUT
     Result : Boolean
EXCEPTIONS
       Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Range_Of SPECIFICATION
     INPUT
The_Domain : Domain,
In_The_Map : Map
     OUTPUT
       Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
generic
      rric
type Domain is private;
type Ranges is private;
Number_Of_Buckets : in Positive;
with function Hash_Of (The_Domain : in Domain) return Positive;
      modified by Tuan Nguyen and Vincent Hong
      date: 8 April 1995
adding procedures to replace functions
      with procedure Hash_Of (The_Domain : in Domain;
                                              Result
                                                               : out Positive);
    end of modication
package Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
is
      type Map is limited private;
                                                          : in Map;
: in out Map);
: in out Map);
: in Domain;
: in Ranges;
      procedure Copy
                                  (From_The_Map : in
                                    To_The_Map
      procedure Clear
procedure Bind
                                  (The_Map : in
(The_Domain : in
And_The_Range : in
      And_The_Man
In_The_Man
procedure Unbind (The_Domain
In_The_Map
                                                           : in out Map);
                                                           : in Domai: in out Map);
      modified by Tuan Nguyen and Vincent Hong
date: 8 April 1995
adding procedures to replace functions
      procedure Is_Equal (Left : in Map;
Right : in Map;
Result : out Boolean);
procedure Extent_Of (The_Map : in Map;
```

```
Result
      procedure Is_Empty
                                           (The Domain : in Domain:
      procedure Is_Bound
                                             In_The_Map : in Map;
Result : out Boolean);
                                           Result : out Boolean)
(The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges);
      procedure Range_Of
-- end of modication
      function Is_Equal (Left : in Map;
function Extent_Of (The_Map : in Map) return Boolean;
function Is_Empty (The_Map : in Map) return Ratural;
function Is_Bound (The_Domain : in Domain;
In_The_Map : in Map) return Boolean;
function Range_Of (The_Domain : in Domain;
In_The_Map : in Map) return Ranges;
              with procedure Process (The_Domain : in Domain;
The_Range : in Ranges;
Continue : out Boolean);
      procedure Iterate (Over_The_Map : in Map);
       Overflow
                                           : exception:
      Domain_Is_Not_Bound : exception;
Multiple_Binding : exception;
private
       type Node;
type Structure is access Node;
type Map is array (Positive range 1 .. Number_Of_Buckets) of
Structure:
end Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FRR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator is
       type Node is
              record
The Domain : Domain;
                                       : Ranges;
: Structure;
                     The_Range
              end record;
       procedure Find (The_Domain In_The_Map
                                                         : in
: in
                                                                            Domain:
                                                                            Map;
Positive;
                                                            : out
                                   The Bucket
          The Bucket out Fostive;
Previous_Node : in out Structure;
Current_Node : in out Structure) is
Temporary_Bucket : Positive :=
(Hash_Of (The_Domain) mod
Number_Of_Buckets) + 1;
              In
The_Bucket := Temporary_Bucket;
Current_Node := In_The_Map(Temporary_Bucket);
while Current_Node /= null loop
    if Current_Node.The_Domain = The_Domain then
                            return;
                     Previous_Node := Current_Node;
Current_Node := Current_Node.Next;
end if;
       end loop;
end Find;
      begin
              for Index in From_The_Map'Range loop
From_Index := From_The_Map(Index);
if From_The_Map(Index) = null then
To_The_Map(Index) := null;
                     else
To_The_Map(Index) := new Node'
(The_Do
                                                                     (The_Domain =>
From Index. The Domain.
                                                                       The Range =>
From Index. The Range.
                                                                                          => null);
                            To_Index := To_The_Map(Index);
                             From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := new Node'
(The_Domain =>
From Index. The Domain,
                                                                       The Range =>
             Next
To_Index := To_Index.Next;
From_Index := From_Index.Next;
end loop;
end if;
end loop;
end loop;
eption
From Index.The_Range,
                                                                                          => null);
       exception
when Storage_Error =>
raise Overflow;
       procedure Clear (The_Map : in out Map) is
       The_Map := Map'(others => null);
end Clear;
       procedure Bind (The_Domain : in Domain;
And_The_Range : in Ranges;
In_The_Map : in out Map) is
The_Bucket : Positive;
              Previous_Node : Structure;
Current_Node : Structure;
       begin
              Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
if Current_Node /= null then
raise Multiple_Binding;
                     In_The_Map(The_Bucket) := new Node'
```

```
(The Domain => The_Domain,
                                                The_Range => And_The_Range,
Next =>
In_The_Map(The_Bucket));
     end if;
exception
    raise Overflow;
end Bind;
          when Storage Error =>
    procedure Unbind (The_Domain : in Domain;
In_The_Map : in out Map) is
The_Bucket : Positive;
Previous_Node : Structure;
Current_Node : Structure;
    begin
          Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
else
Previous_Node.Next := Current_Node.Next;
end if;
     exception when Constraint_Error =>
              raise Domain_Is_Not_Bound;
     end Unbind;
    modified by Tuan Nguyen and Vincent Hong
    date: 8 April 1995
adding procedures to replace functions
    Result := Is_Equal(Left,Right);
    begin
    Result := Extent_Of(The_Map);
end Extent_Of;
    Result := Is_Empty(The_Map);
end Is_Empty;
    Result := Is_Bound(The_Domain,In_The_Map); end Is_Bound;
    procedure Range_Of (The_Domain : in Domain;
In_The_Map : in Map;
Result : out Ranges) is
    Result := Range_Of(The_Domain,In_The_Map);
end Range_Of;
-- end of modification
    begin
         for Index in Left'Range loop
   if (Left(Index) = null) xor (Right(Index) = null) then
        return False;
              else
                  Left_Index := Left(Index);
Left_Count := 0;
while Left_Index /= null loop
    Right_Index := Right(Index);
    while Right_Index /= null loop
    if (Left_Index.The_Domain =
        Right_Index.The_Domain) then
    exit:
                                 exit;
                            Right_Index := Right_Index.Next;
end if;
                       end loop;
if Left_Index.The_Range /= Right_Index.The_Range
then
                            return False:
                       else
   Left_Index := Left_Index.Next;
   Left_Count := Left_Count + 1;
```

```
Right_Index := Right(Index);
Right_Count := 0;
While Right_Index /= null loop
Right_Index := Right_Index.Next;
Right_Count := Right_Count + 1;
end loop;
if Left_Count /= Right_Count then
return False;
end if;
end if;
end oop;
return True;
exception
When Constraint_Error =>
return False;
end Is_Equal;

function Extent_Of (The_Map : in Map) return Natural is
Count : Natural := 0;
Temporary_Node := Structure;

begin
for Index in The_Map'Range loop
Temporary_Node := The_Map(Index);
while Temporary_Node /= null loop
Count := Count + 1;
Temporary_Node := Temporary_Node.Next;
end loop;
end loop;
return Count;
end Extent_Of;

function Is_Empty (The_Map : in Map) return Boolean is
begin
return (The_Map = Map'(others => null));
end Is_Empty;

function Is_Bound (The_Domain : in Domain;
In_The_Map : in Map) return Boolean is
The_Bucket : Positive;
Previous_Node : Structure;
Current_Node : Structure;
begin
Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
return (Current_Node /= null);
end Is_Bound;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
TYPE Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator SPECIFICATION GENERIC
      Domain : PRIVATE TYPE,
     Domain : PRIVATE_TYPE,
Ranges : PRIVATE_TYPE,
Hash_Of : FUNCTION[The_Domain : Domain, RETURN : Positive],
Hash_Of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positive]]
OPERATOR Copy
SPECIFICATION
      INPUT
        From_The_Map : Map,
To_The_Map : Map
      OUTPUT
     To_The_Map : Map
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Clear
   SPECIFICATION
        The_Map : Map
      OUTPIE
     The_Map : Map
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Bind
   SPECIFICATION
        The_Domain : Domain,
And_The_Range : Ranges,
      In_The_Map : Map
OUTPUT
In_The_Map : Map
      EXCEPTIONS
         Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR Unbind
   SPECIFICATION
INPUT
        The Domain : Domain,
      In_The_Map : Map
OUTPUT
In_The_Map : Map
      EXCEPTIONS
         Overflow, Domain_Is_Not_Bound, Multiple_Binding
   OPERATOR IS_Equal SPECIFICATION
     INPUT
Left : Map
      Right : Map
OUTPUT
Result : Boolean
EXCEPTIONS
```

```
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Extent_Of SPECIFICATION
     INPUT
The_Map : Map
OUTPUT
Result : Natural
     EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Is_Empty
SPECIFICATION
     INPUT
     The_Map : Map
OUTPUT
Result : Boolean
     EXCEPTIONS
Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END
  OPERATOR Is_Bound SPECIFICATION
    The Domain : Domain,
In_The Map : Map
OUTPUT
Result : Boolean
EXCEPTIONS
CHEETICAN DOMAIN IS I
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
  OPERATOR Range_Of SPECIFICATION
     TNPDT
     The_Domain : Domain,
In_The_Map : Map
OUTPUT
        Result : Ranges
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   EMD
  OPERATOR Iterate SPECIFICATION
      GENERIC
Process : PROCEDURE[The_Domain : in[t : Domain], The_Range : in[t : Ranges], Continue : out[t : Boolean]]
      INPUT
      Over_The_Map : Map
EXCEPTIONS
        Overflow, Domain_Is_Not_Bound, Multiple_Binding
   ENTO
IMPLEMENTATION ADA
MMD_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
END
```

QUEUES OBJ3 SPECIFICATION

```
obj QUEUE[X :: TRIV] is sort Queue .
    protecting NAT .
    subsorts NzNat < Nat .

*** constructors

    op create : -> Queue .
    op copy : Queue Queue -> Queue .
    op add : Elt Queue -> Queue .
    op pop : Queue -> Queue .
    op removeitem : Queue NzNat -> Queue .

*** accessors

    op isequal : Queue NzNat -> Queue .
    op lengthof : Queue -> Nat .
    op isempty : Queue -> Bool .
    op isempty : Queue -> Bool .
    op positionof : Elt Queue -> Nat .

*** exceptions

    op overflow : -> Queue .
    op underflow : -> Queue .
    op underflow : -> Elt .
    op positionerror : -> Nat .

*** variables declarations

    var Q Q1 : Queue .
```

QUEUES PROFILE CODES

OPERATORS	SIGNATURES	PROFILE CODES
COPY	A B -> B	3211
CLEAR	A -> A	2201
ADD	A B -> B	3211
POP	A -> A	2201
REMOVE_ITEM	A B -> A	3211
IS_EQUAL	A B -> C	330
LENGTH_OF	A -> B	220
IS_EMPTY	A -> B	220
FRONT_OF	A -> B	220
POSITION_OF	A -> B	220

SET OF PROFILE: {3211,2201,330,220}

QUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
type Item is private;
package Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator
is
         type Queue(The_Size : Positive) is limited private;
                                                                                       : in Queue;
: in out Queue);
: in out Queue;
: in Positive);
                                                      (From_The_Queue
        procedure Copy
                                                       To The Oueue
        procedure Clear (The_Queue procedure Add (The_Item To_The_Queue procedure Pop (The_Queue procedure Remove_Item (From_The_Queue At_The_Position
       modified by Tuan Nguyen replacing functions with procedures
                                                                                  : in Queue;
: in Queue;
: out Boolean);
: in Queue;
         procedure Is_Equal
                                                      (Left
                                                     Right
Result
(The_Queue
Result
        procedure Length_Of
                                                                                      out Natural);
        procedure Is_Empty (The_Queue Result procedure Front_Of (The_Queue Result procedure Fosition_Of (The_Item In_The_Queue
                                                                                     in Queue;
out Boolean);
in Queue;
                                                                                      Item):
                                                                                  : in Item;
: in Queue;
```

OUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52 227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body
Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator is
      procedure Copy (From_The_Queue : in Queue;
    To_The_Queue : in out Queue) is
             if From_The_Queue.The_Back > To_The_Queue.The_Size then
  raise Overflow;
             raise Overriow;
elsif From_The_Queue.The_Back = 0 then
   To_The_Queue.The_Back := 0;
             else
                   To_The_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
   From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
To_The_Queue.The_Back := From_The_Queue.The_Back;
             end if:
      end Copy;
      procedure Clear (The_Queue : in out Queue) is begin
             The_Queue.The_Back := 0;
      end Clear;
      Int
To_The_Queue.The_Items(To_The_Queue.The_Back + 1) := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
      exception
when Constraint_Error =>
raise Overflow;
end Add;
      procedure Pop (The_Queue : in out Queue) is begin
            if The_Queue.The_Back = 0 then
            raise Underflow;
elsif The_Queue.The_Back = 1 then
The_Queue.The_Back := 0;
                   The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
    The_Queue.The_Items(2 .. The_Queue.The_Back);
The_Queue.The_Back := The_Queue.The_Back - 1;
             end if:
      end Pop;
     begin
if From_The_Queue.The_Back < At_The_Position then
    raise Position_Error;
elsif From_The_Queue.The_Back /= At_The_Position then</pre>
                   From_The_Queue.The_Items
(At_The_Position .. (From_The_Queue.The_Back - 1)) :=
From_The_Queue.The_Items
((At_The_Position + 1) .. From_The_Queue.The_Back);
             From_The_Queue.The_Back := From_The_Queue.The_Back - 1;
      end Remove Item:
    modified by Tuan Nguyen replacing functions with procedures
                                                                  : in Queue;
: in Queue;
: out Boolean) is
                                           Left
      procedure Is_Equal
                                             Result
      Result := Is_Equal(Left,Right);
end Is_Equal;
```

```
procedure Length Of (The Oueue
                                                        : in Queue;
                                                        : out Natural) is
           Result := Length_Of(The_Queue);
     end Length_Of;
     procedure Is_Empty
                                                        : out Boolean) is
                                     Result
     Result := Is_Empty(The_Queue);
end Is_Empty;
                                    (The_Queue
Result
                                                        : in Queue;
: Item) is
     procedure Front_Of
     Result := Front_Of(The_Queue);
end Front_Of;
     procedure Position_Of (The_Item
                                                       : in Item:
                                     In_The_Queue : in Queue;
Result : out Natural) is
     Result := Position_Of(The_Item,In_The_Queue);
end Position_Of;
     end of modification
     function Is_Equal (Left : in Queue;
Right : in Queue) return Boolean is
          if Left.The_Back /= Right.The_Back then
    return False;
           else
   .. Lea
...The_Items

return False;
end if;
end loop;
return True;
end if;
end Is_Equal;
function '
begir
                for Index in 1 .. Left.The_Back loop
if Left.The_Items(Index) /= Right.The_Items(Index)
then
     function Length_Of (The_Queue : in Queue) return Natural is
     begin
    return The_Queue.The_Back;
end Length_Of;
     function Is_Empty (The_Queue : in Queue) return Boolean is
     begin
return (The_Queue.The_Back = 0);
end Is_Empty;
     function Front_Of (The_Queue : in Queue) return Item is
     begin
  if The_Queue.The_Back = 0 then
    raise Underflow;
  else
          return The_Queue.The_Items(1);
end if;
     end Front_Of;
     function Position_Of (The_Item : in Item;
In_The_Queue : in Queue) return Natural is
          for Index in 1 .. In_The_Queue.The_Back loop
    if In_The_Queue.The_Items(Index) = The_Item then
        return Index;
end if;
end loop;
          return 0;
     end Position_Of;
     begin
   for The_Iterator in 1 .. Over_The_Queue.The_Back loop
        Process(Over_The_Queue.The_Items(The_Iterator), Continue);
        exit when not Continue;
end loop;
end Iterate;
```

end Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator;

QUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
TYPE Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator SPECIFICATION
  PECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR CODY
SPECIFICATION
INPUT
From_The_Queue: Queue,
To_The_Queue: Queue
OUTPUT
       OUTPUT
To_The_Queue : Queue
       EXCEPTIONS
          Overflow, Underflow, Position_Error
   OPERATOR Clear SPECIFICATION
       INPUT
      INPUT
The_Queue : Queue
OUTFUT
The_Queue : Queue
EXCEPTIONS
          Overflow, Underflow, Position_Error
   OPERATOR Add
   OPERATOR Add
SPECIFICATION
INPUT
The_Item : Item,
To_The_Queue : Queue
       TO_INE_Queue : Queue
OUTPUT
TO_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
    END
    OPERATOR Pop
SPECIFICATION
       INPUT
       The_Queue : Queue
OUTPUT
The_Queue : Queue
EXCEPTIONS
           Overflow, Underflow, Position_Error
    OPERATOR Remove_Item
    SPECIFICATION
INPUT
From_The_Queue : Queue,
       At_The_Position : Positive
OUTPUT
From_The_Queue : Queue
EXCEPTIONS
           Overflow, Underflow, Position_Error
    OPERATOR Is Equal
    SPECIFICATION
INPUT
Left : Queue,
Right : Queue
```

```
OUTPUT
     Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Length_Of
SPECIFICATION
INFUT
The_Queue : Queue
OUTFUT
Result : Natural
EXCEPTIONS
Overflow, Underflow, Position_Error
  END
  OPERATOR IS_Empty
SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
     OUTPUT

Result : Boolean

EXCEPTIONS

Overflow, Underflow, Position_Error
  OPERATOR Front_Of SPECIFICATION
     INPUT
     INPUT
The_Queue : Queue
OUTPUT
Result : Item
EXCEPTIONS
         Overflow, Underflow, Position_Error
   EMD
  OPERATOR Position_Of SPECIFICATION
      INPUT
        The_Item : Item,
In_The_Queue : Queue
     OUTPUT
        Result : Natural
  EXCEPTIONS
Overflow, Underflow, Position_Error
END
  OPERATOR Iterate
SPECIFICATION
      GENERIC
         Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
      INPUT
      Over_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
  END
END
IMPLEMENTATION ADA
Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator
END
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
generic
generic
    type Item is private;
package
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator is
       type Queue is limited private;
                                                                          : in Queue;
: in out Queue);
                                             (From_The_Queue
To_The_Queue
(The_Queue
(The_Item
To_The_Queue
      procedure Copy
      procedure Clear
procedure Add
                                             (The_Queue :
(From_The_Queue :
At_The_Position :
      procedure Pop
procedure Remove_Item
       procedure Is_Equal
                                              (Left
                                                                      : in Oueue;
                                                                        in Queue;
out Boolean);
                                               Right
                                               Result
      procedure Length_Of
                                              (The_Queue
Result
                                                                        in Queue;
out Natural);
                                             (The_Queue
Result
(The_Queue
                                                                        in Queue;
       procedure Is_Empty
                                                                        out Boolean);
in Queue;
      procedure Front_Of
                                               Result
                                                                        Item):
      procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
```

```
Result : out Natural);

-- end of modification

function Is_Equal (Left : in Queue;
    Right : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty (The_Queue : in Queue) return Boolean;
    function Front_Of (The_Queue : in Queue) return Boolean;
    function Fosition_Of (The_Queue : in Queue) return Item;
    function Position_Of (The_Item : in Item;
        In_The_Queue : in Queue) return Natural;

Overflow : exception;
    Underflow : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Structure is access Node;
    type Queue is
        record
        The_Fack : Structure;
        end record;

end

Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
   "Restricted Rights Legend"
Use, duplication, or disclosure is subject to restrictions as set forth in subdivision (b) (3) (ii) of the rights in Technical Data and Computer Software Clause of FAR 52.227-7013. Manufacturer: Wizard software, 2171 S. Parfet Court, Lakewood, Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator
      type Node is record
                 The Item : Item;
                 Next
                               : Structure;
            end record;
      procedure Free (The_Node : in out Node) is
begin
     null;
end Free;
     Structure) is
      begin
           The Node Next := To Next:
      function Next_Of (The_Node : in Node) return Structure is
      return The_Node.Next;
end Next_Of;
      package Node_Manager is new Storage_Manager_Sequential
                                                      (Item
     end if;
     end 11,
exception
when Storage_Error =>
raise Overflow;
      procedure Clear (The_Queue : in out Queue) is begin
            Node_Manager.Free(The_Queue.The_Front);
The_Queue.The_Back := null;
      end Clear:
      procedure Add (The_Item : in Item;
To_The_Queue : in out Queue) is
            n
if To_The_Queue.The_Front = null then
To_The_Queue.The_Front := Node_Manager.New_Item;
To_The_Queue.The_Front.The_Item := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Front;
            else
To_The_Queue.The_Back.Next := Node_Manager.New_Item;
To_The_Queue.The_Back.Next.The_Item:= The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
            end if:
       exception
when Storage_Error =>
raise Overflow;
      procedure Pop (The_Queue : in out Queue) is
   Temporary_Node : Structure;
```

```
Temporary_Node.Next := null;
Node_Manager.Free(Temporary_Node);
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
      exception when Constraint_Error =>
                 raise Underflow;
     procedure Remove_Item (From_The_Queue : in out Queue;
At_The_Position : in Positi
Count : Natural := 1;
Previous : Structure;
                                                                              Positive) is
            Index : Structure := From_The_Queue.The_Front;
     begin
while Index /= null loop
if Count = At_The_Position then
                       Count := Count + 1;
                 Previous := Index;
Index := Index.Next;
end if;
            end loop;
if Index = null then
    raise Position_Error;
elsif Previous = null then
                 From_The_Queue.The_Front := Index.Next;
            Previous.Next := Index.Next;
end if;
           end if;
if From_The_Queue.The_Back = Index then
    From_The_Queue.The_Back := Previous;
end if;
Index.Next := null;
Node_Manager.Free(Index);
      end Remove_Item;
-- modified by Tuan Nguyen
-- replacing functions with procedures
                                      (Left
                                                           : in Queue;
: in Queue;
: out Boolean) is
      procedure Is_Equal
      begin
            Result := Is_Equal(Left, Right);
      end Is_Equal;
                                                            : in Queue;
: out Natural) is
      procedure Length_Of
                                        Result
      Result := Length_Of(The_Queue);
end Length_Of;
      procedure Is_Empty
                                       (The Oueue
                                                            : in Oueue;
                                                            : out Boolean) is
      Result := Is_Empty(The_Queue);
end Is_Empty;
                                                            : in Queue;
: Item) is
      procedure Front_Of
                                        Result
      Result := Front_Of(The_Queue);
end Front_Of;
      Result := Position_Of(The_Item,In_The_Queue);
end Position_Of;
-- end of modification
      function Is_Equal (Left : in Queue;
    Right : in Queue) return Boolean is
    Left_Index : Structure := Left_The_Front;
    Right_Index : Structure := Right_The_Front;
      begin
while Left_Index /= null loop
   if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
                  else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
                  end if;
            end loop;
return (Right_Index = null);
      exception
            when Constraint_Error =>
      return False;
end Is_Equal;
      function Length_Of (The_Queue : in Queue) return Natural is
```

```
Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;

begin
   while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
   end loop;
   return Count;
end Length_Of;

function Is_Empty (The_Queue : in Queue) return Boolean is
begin
   return (The_Queue.The_Front = null);
end Is_Empty;

function Front_Of (The_Queue : in Queue) return Item is
begin
   return The_Queue.The_Front.The_Item;
exception
   when Constraint_Error =>
        raise Underflow;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator SPECIFICATION
  GENERIC
Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
     INPUT
From_The_Queue : Queue,
To_The_Queue : Queue
     OUTPUT
To_The_Queue : Queue
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Clear
SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
         The_Queue : Queue
      EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Add
SPECIFICATION
     The_Item : Item,
To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Pop
SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
        The_Queue : Queue
     EXCEPTIONS
Overflow, Underflow, Position_Error
  OPERATOR Remove_Item SPECIFICATION
     INPUT
     INPUT
From The Queue : Queue,
At The Position : Positive
OUTPUT
From The Queue : Queue
EXCEPTIONS
        Overflow, Underflow, Position_Error
```

```
OPERATOR Is_Equal
   SPECIFICATION
INPUT
Left: Queue,
     Right : Queue
OUTPUT
Result : Boolean
     EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Length_Of SPECIFICATION
     INPUT
        The_Queue : Queue
     OUTPUT

Result : Natural

EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR IS_Empty SPECIFICATION
    INPUT
The_Queue : Queue
OUTPUT
     Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Front_Of SPECIFICATION
    INPUT
        The_Queue : Queue
    OUTPUT
Result : Item
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Position_Of SPECIFICATION
    PECIFICAL
INPUT
The_Item : Item,
In_The_Queue : Queue
    OUTPUT
Result : Natural
EXCEPTIONS
       Overflow, Underflow, Position_Error
  END
INFORMATION ADA
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator
END
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
generic
       type Item is private;
package
package
Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator is
      type Queue(The_Size : Positive) is limited private;
      procedure Copy (From_The_Queue : in
     procedure Copy (From_The_Queue : in Queue);

procedure Clear (The_Queue : in out Queue);

procedure Add (The_Item : in out Queue);

procedure Pop (The_Queue : in out Queue);

procedure Pop (The_Queue : in out Queue);
      procedure Pop (The_Queue
     modified by Tuan Nguyen replacing functions with procedures
                                                                   : in Queue;
: in Queue;
: out Boolean);
: in Queue;
: out Natural);
      procedure Is_Equal
                                            Right
Result
(The_Queue
Result
      procedure Length_Of
                                           (The_Queue
Result
(The_Queue
                                                                   : in Queue;
: out Boolean);
: in Queue;
      procedure Is_Empty
      procedure Front_Of
                                             Result
                                                                    : Item);
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
    Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator
      procedure Copy (From_The_Queue : in Queue;
To_The_Queue : in out Queue) is
              To_The_Queue.The_Back := 0;
              ro_Tne_Queue.Tne_Back := 0;
else
   To_Tne_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
        From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
   To_The_Queue.The_Back := From_The_Queue.The_Back;
end if;
       end Copy;
       procedure Clear (The_Queue : in out Queue) is
begin
              The_Queue.The_Back := 0;
      procedure Add (The_Item : in Item;
To_The_Queue : in out Queue) is
              To_The_Queue.The_Items(To_The_Queue.The_Back + 1) := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
      when Constraint_Error => raise Overflow; end Add;
       exception
       procedure Pop (The_Queue : in out Queue) is
        oroccute for the back = 0 then raise Underflow; elsif The_Queue.The_Back = 1 then The_Queue.The_Back := 0;
      else
    The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
          The_Queue.The_Items(2 .. The_Queue.The_Back);
    The_Queue.The_Back := The_Queue.The_Back - 1;
end if;
end Fop;
      modified by Tuan Nguyen replacing functions with procedures
                                                                         : in Queue;
: in Queue;
       procedure Is_Equal
                                                 Right
                                                 Result
                                                                         : out Boolean) is
       begin
              Result := Is_Equal(Left, Right);
```

```
end Is Equal:
    procedure Length_Of (The_Queue
                                          : in Queue;
                            Result
                                          : out Natural) is
    Result := Length_Of(The_Queue);
end Length_Of;
                           (The_Queue
Result
    procedure Is_Empty
                                          : in Queue;
: out Boolean) is
        Result := Is_Empty(The_Queue);
    end Is_Empty;
                                          : in Queue;
: Item) is
   procedure Front_Of
                           (The Oueue
    Result := Front_Of(The_Queue);
end Front_Of;
   end of modification
    if Left.The_Back /= Right.The_Back then
        return False;
            for Index in 1 .. Left.The_Back loop
if Left.The_Items(Index) /= Right.The_Items(Index)
  .. Lei
..The_Items

return False;
end if;
end loop;
return True;
end if;
end Is_Equal;

function
begi-
    function Length_Of (The_Queue : in Queue) return Natural is
       return The Oueue. The Back;
    end Length_Of;
    function Is_Empty (The_Queue : in Queue) return Boolean is
         return (The_Queue.The_Back = 0);
    end Is_Empty;
    function Front_Of (The_Queue : in Queue) return Item is
    begin
if The_Queue.The_Back = 0 then
raise Underflow;
        else
    return The_Queue.The_Items(1);
end if;
end Front_Of;
   procedure Iterate (Over_The_Queue : in Queue) is
        Continue : Boolean;
   end Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
{\tt TYPE\ Queue\_Nonpriority\_Nonbalking\_Sequential\_Bounded\_Managed\_Iterator\ SPECIFICATION \\
   GENERIC
   GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
From_The_Queue : Queue,
To_The_Queue : Queue
      OUTPUT
      To_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow
   END
   OPERATOR Clear
SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
      The_Queue : Queue EXCEPTIONS
         Overflow, Underflow
   END
   OPERATOR Add
   SPECIFICATION
INPUT
      INFUT
The_Item : Item,
To_The_Queue : Queue
OUTPUT
         To_The_Queue : Queue
      EXCEPTIONS
Overflow, Underflow
   END
   OPERATOR POP
SPECIFICATION
      INPUT
         The_Queue : Queue
      OUTPUT
The_Queue : Queue
EXCEPTIONS
         Overflow, Underflow
   OPERATOR Is_Equal SPECIFICATION
      INPUT
Left : Queue,
         Right : Queue
```

```
Result : Boolean
     EXCEPTIONS
        Overflow, Underflow
  OPERATOR Length_Of SPECIFICATION
     INPUT
       The_Queue : Queue
     OUTPUT
Result : Natural
EXCEPTIONS
  EACEPTIONS
Overflow, Underflow
END
  OPERATOR IS_Empty
SPECIFICATION
INPUT
     INPUT
The_Queue : Queue
OUTPUT
Result : Boolean
  Overflow, Underflow
  OPERATOR Front_Of SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
     Result : Item
EXCEPTIONS
       Overflow, Underflow
  OPERATOR Iterate SPECIFICATION
     GENERIC
Process: PROCEDURE[The_Item : in[t : Item], Continue : out[t : Boolean])
INPUT
      Over_The_Queue : Queue
  EXCEPTIONS
Overflow, Underflow
END
END
IMPLEMENTATION ADA
Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
procedure Front_Of (The_Queue : in Queue;
Result : Item);

-- end of modification

function Is_Equal (Left : in Queue)
Function Length_Of (The_Queue : in Queue) return Boolean;
function Is_Empty (The_Queue : in Queue) return Natural;
function Is_Empty (The_Queue : in Queue) return Boolean;
function Front_Of (The_Queue : in Queue) return Item;

Overflow : exception;
Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
    record
        The_Front : Structure;
        the Back : Structure;
        end record;

end
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;
```

OUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"

- Use, duplication, or disclosure is subject to

- restrictions as set forth in subdivision (b) (3) (ii)

- of the rights in Technical Data and Computer

- Software Clause of FAR 52.227-7013. Manufacturer:

- Wizard software, 2171 S. Parfet Court, Lakewood,

- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
        type Node is
                record
                        The_Item : Item;
                                           : Structure;
                end record;
       procedure Free (The_Node : in out Node) is begin
       null;
end Free;
       procedure Set_Next (The_Node : in out Node;
To_Next : in Struc
                                                                                       Structure) is
       The_Node.Next := To_Next;
end Set_Next;
        function Next_Of (The_Node : in Node) return Structure is
       begin return The_Node.Next;
end Next_Of;
       age_manager_sequential (Item => Node, Pointer => Structure, Free => Free, Set_Pointer => Set_Next, Pointer_Of => Next_Of);
      in

Node_Manager.Free(To_The_Queue.The_Front);

To_The_Queue.The_Back := null;

if From_The_Queue.The_Front := Node_Manager.New_Item;

To_The_Queue.The_Front := Node_Manager.New_Item;

To_The_Queue.The_Back := To_The_Queue.The_Front;

To_The_Queue.The_Front.The_Item := From_Index.The_Item;

To_Index := To_The_Queue.The_Front;

From_Index := From_Index.Next;

while From_Index /= null loop

To_Index.Next := Node_Manager.New_Item;

To_Index.Next := Node_Manager.New_Item;

To_Index.Sext.The_Item := From_Index.The_Item;

To_Index := To_Index.Next;

From_Index := From_Index.Next;

To_The_Queue.The_Back := To_Index;

end loop;

end loop;
                       end loop;
               end if;
      exception
when Storage_Error =>
                      raise Overflow;
       end Copy;
      procedure Clear (The_Queue : in out Queue) is
              In
Node_Manager.Free(The_Queue.The_Front);
The_Queue.The_Back := null;
      procedure Add (The_Item : in Item;
To_The_Queue : in out Queue) is
               if To_The_Queue.The_Front = null then
                       To_The_Queue.The_Front := Node_Manager.New_Item;
To_The_Queue.The_Front.The_Item := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Front;
              To_The_Queue.The_Back.Next := Node_Manager.New_Item;
To_The_Queue.The_Back.Next.The_Item:= The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
end if;
```

```
when Storage Error =>
      procedure Fop (The_Queue : in out Queue) is
   Temporary_Node : Structure;
      begin
           Temporary_Node := The_Queue.The_Front;
The_Queue.The_Front := The_Queue.The_Front.Next;
Temporary_Node.Next := null;
           Node_Manager.Free(Temporary_Node);
if The_Queue.The_Front = null then
The_Queue.The_Back := null;
end if;
      exception
when Constraint_Error =>
raise Underflow;
     end Pop:
-- modified by Tuan Nguyen
-- replacing functions with procedures
     procedure Is_Equal
                                   (Left
                                                      : in Oueue:
                                    Right
                                                      · in Oueue:
                                                      : out Boolean) is
           Result := Is_Equal(Left,Right);
     end Is_Equal;
     procedure Length Of
                                   (The_Queue
Result
                                                      : in Oueue:
                                                      : out Natural) is
          Result := Length_Of(The_Queue);
     end Length_Of;
     procedure Is_Empty
                                   (The_Queue
Result
           Result := Is_Empty(The_Queue);
     end Is_Empty;
     procedure Front_Of
                                   (The_Queue
                                                      : in Queue;
: Item) is
                                    Result
    begin
    Result := Front_Of(The_Queue);
end Front_Of;
-- end of modification
    .n
while Left_Index /= null loop
if Left_Index.The_Item /= Right_Index.The_Item then
                    return False;
                else
   Left_Index := Left_Index.Next;
   Right_Index := Right_Index.Next;
              end if:
          end loop;
return (Right_Index = null);
     exception
          when Constraint_Error =>
    return False;
end Is_Equal;
    function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
          while Index /= null loop
Count := Count + 1;
Index := Index.Next;
          end loop;
    return Count;
end Length_Of;
     function Is_Empty (The_Queue : in Queue) return Boolean is
         return (The_Oueue.The_Front = null);
    end Is_Empty;
    function Front_Of (The_Queue : in Queue) return Item is
          return The_Queue.The_Front.The_Item;
    exception
when Constraint_Error =>
               raise Underflow:
    end Front_Of;
```

Oueue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
TYPE
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
INPUT
From_The_Queue: Queue
OUTPUT
To_The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END
OPERATOR Clear
SPECIFICATION
INPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Add
SPECIFICATION
INPUT
The_Item: Item,
To_The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Add
SPECIFICATION
INPUT
The_Item : Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
To_The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END
```

QUEUE PRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
generic
       type Item is private;
type Priority is limited private;
with function Priority_Of (The_Item : in Item)
with function "<=" (Left : in Priority;
Right : in Priority) return Boolean;
package Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator is
       type Queue(The_Size : Positive) is limited private;
                                                  (From The Oueue : in
       procedure Copy
                                                                                                  Queue;
                                                                                 : in out Queue);
: in out Queue);
: in Item;
                                                  To_The_Queue
(The_Queue
(The_Item
       procedure Clear
procedure Add
       procedure Add (TRE_IEEM : in Item;
TO_The_Queue : in out Queue);
procedure Pop (The_Queue : in out Queue);
procedure Remove_Item (From_The_Queue : in out Queue;
At_The_Position : in Positive);
     modified by Tuan Nguyen replacing functions with procedures
                                                                           : in Queue;
: in Queue;
: out Boolean);
: in Queue;
       procedure Is Equal
                                                   Right
                                                  Result
(The_Queue
       procedure Length_Of
                                                                           : in Queue;
: out Natural);
: in Queue;
: out Boolean);
: in Queue;
                                                   Result
       procedure Is_Empty
                                                  (The_Queue
Result
                                                 (The_Queue
Result
       procedure Front_Of
                                                                            : Item);
```

OUEUE PRIORITY BALKING SEOUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch -- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of PAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator is
      if From_The_Queue.The_Back > To_The_Queue.The_Size then
    raise Overflow;
elsif From_The_Queue.The_Back = 0 then
                    To_The_Queue.The_Back := 0;
             else
   To_The_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
        From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
   To_The_Queue.The_Back := From_The_Queue.The_Back;
};
      end if;
end Copy;
      procedure Clear (The_Queue : in out Queue) is
begin
      The_Queue.The_Back := 0;
end Clear;
      in
if To_The_Queue.The_Back = 0 then
    To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
The Item:
                   To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
             end loop;
if Index > To_The_Queue.The_Back then
   To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
The_Item:
                          To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
                    else
                          To_The_Queue.The_Items
((Index + 1) .. (To_The_Queue.The_Back + 1)) :=
To_The_Queue.The_Items(Index ..
            To_The_Queue.The_Items(Index) := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
end if;
end if;
To The Oueue. The Back):
      exception
when Constraint_Error =>
raise Overflow;
end Add;
       procedure Pop (The_Queue : in out Queue) is
       begin
if The_Queue.The_Back = 0 then
             raise Underflow;
elsif The_Queue.The_Back = 1 then
The_Queue.The_Back := 0;
                   The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
   The_Queue.The_Items(2 .. The_Queue.The_Back);
The_Queue.The_Back := The_Queue.The_Back - 1;
             end if:
      end Pop;
      in
if From_The_Queue.The_Back < At_The_Position then
    raise Position_Error;
elsif From_The_Queue.The_Back /= At_The_Position then
    From_The_Queue.The_Items
    (At_The_Position . (From_The_Queue.The_Back - 1)) :=
        From_The_Queue.The_Items
        ((At_The_Position + 1) .. From_The_Queue.The_Back);</pre>
       From_The_Queue.The_Back := From_The_Queue.The_Back - 1; end Remove_Item;
```

```
modified by Tuan Nguyen replacing functions with procedures
     procedure Is Equal
                                  (Left
                                                    : in Queue;
: in Queue;
                                   Right
Result
                                                     : out Boolean) is
     begin
          Result := Is_Equal(Left,Right);
     end Is_Equal;
     procedure Length_Of (The_Queue Result
                                                    : in Queue;
: out Natural) is
     begin
     Result := Length_Of(The_Queue);
end Length_Of;
                                  (The_Queue
Result
     procedure Is_Empty
                                                    : in Queue;
                                                    : out Boolean) is
     begin
          Result := Is_Empty(The_Queue);
     end Is_Empty;
                                                    : in Queue;
: Item) is
     procedure Front Of
                                  (The Oueue
     begin
          Result := Front Of (The Oueue);
     end Front_Of;
    Result := Position_Of(The_Item,In_The_Queue);
end Position_Of;
    end of modification
     function Is_Equal (Left : in Queue;
Right : in Queue) return Boolean is
          if Left.The_Back /= Right.The_Back then
    return False;
          else
               for Index in 1 .. Left.The_Back loop
if Left.The_Items(Index) /= Right.The_Items(Index)
         return False;
end if;
end loop;
return True;
end if;
Is F-
then
     end Is_Equal;
     function Length_Of (The_Queue : in Queue) return Natural is
     return The_Queue.The_Back;
end Length_Of;
     function Is_Empty (The_Queue : in Queue) return Boolean is
     return (The_Queue.The_Back = 0);
end Is_Empty;
     function Front_Of (The_Queue : in Queue) return Item is
     begin
if The_Queue.The_Back = 0 then
raise Underflow;
          else
          return The_Queue.The_Items(1);
end if;
     end Front Of:
     function Position_Of (The_Item : in Item;
In_The_Queue : in Queue) return Natural is
          if In_The_Queue.The_Back loop
   if In_The_Queue.The_Items(Index) = The_Item then
        return Index;
   end if;
     end loop;
return 0;
end Position_Of;
    procedure Iterate (Over_The_Queue : in Queue) is
   Continue : Boolean;
          for The_Iterator in 1 .. Over_The_Queue.The_Back loop
    Process(Over_The_Queue.The_Items(The_Iterator), Continue);
    exit when not Continue;
     end loop;
end Iterate;
```

end Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator;

QUEUE PRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
TYPE Oueue_Priority_Balking_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  PECTICATION
GENERIC
Item : PRIVATE_TYPE,
Priority : PRIVATE_TYPE,
Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :</pre>
Boolean]
 oolean]
OPERATOR Copy
SPECIFICATION
INFUT
From_The_Queue : Queue,
To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Clear
  SPECIFICATION
     INPUT
        The_Queue : Queue
     OUTPUT
The_Queue : Queue
     EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Add
SPECIFICATION
     INPUT
The_Item : Item,
To_The_Queue : Queue
     To_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
  END
  OPERATOR Pop
SPECIFICATION
     INPUT
        The_Queue : Queue
     OUTPUT
The_Queue : Queue
     EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Remove_Item
  SPECIFICATION
INPUT
From_The_Queue : Queue,
     At_The_Position : Positive OUTPUT
     From_The_Queue : Queue
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Is_Equal
SPECIFICATION
INPUT
```

```
Left : Queue,
Right : Queue
OUTPUT
        Result : Boolean
     EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Length_Of
SPECIFICATION
INFUT
The_Queue : Queue
OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Is_Empty
  SPECIFICATION
INPUT
        The_Queue : Queue
       Result : Boolean
     EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Front_Of SPECIFICATION
     INPUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Position_Of
  OPERATOR POSITION_OF
SPECIFICATION
INPUT
The_Item : Item,
In_The_Queue : Queue
     OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Iterate SPECIFICATION
     GENERIC
       Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean 11
     INPUT
     Over_The_Queue : Queue EXCEPTIONS
       Overflow, Underflow, Position_Error
  EMD
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
generic
      type Item is private;
type Priority is limited private;
with function Priority_Of (The_Item : in Item)
Priority;
with function "<="
                                                   (Left
                                                                   : in Priority;
: in Priority) return Boolean;
package
Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator is
       type Queue is limited private;
                                            (From_The_Queue
To_The_Queue
(The_Queue
(The_Item
To_The_Queue
(The_Queue
                                                                        : in Queue;

: in out Queue);

: in out Queue);

: in Item;

: in out Queue);

: in out Queue);

: in out Queue;
      procedure Clear
procedure Add
      procedure Pop
procedure Remove_Item
                                            (From The Oueue
                                             At_The_Position
                                                                           in
                                                                                        Positive):
     modified by Tuan Nguyen replacing functions with procedures
                                                                       in Queue;
in Queue;
out Boolean);
      procedure Is_Equal
                                             Right
                                            Result
(The_Queue
Result
      procedure Length_Of
                                                                       in Queue;
out Natural);
                                            (The Oueue
      procedure Is_Empty
                                                                    : in Queue;
```

```
: out Boolean);
: in Queue
       procedure Front_Of (The_Queue Result )
procedure Position_Of (The_Item In_The_Queue Result
                                                                                  in Queue;
Item);
in Item;
                                                                                  in Queue;
out Natural);
       end of modification
        function Is_Equal
                                                  (Left
                                                                             : in Queue;
                                                                               in Queue; in Queue) return Boolean; in Queue) return Matural; in Queue) return Boolean; in Queue) return Item; in Item;
       function Length_Of (The_Queue function Is_mmpty (The_Queue function Front_Of (The_Queue function Position_Of (The_Item_In_The_Queue
                                                                                                    return Natural;
       Overflow : exception;
Underflow : exception;
Position_Error : exception;
private
       type Node;
type Structure is access Node;
       type Queue is
record
The_Front : Structure;
The_Back : Structure;
               end record:
end Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator;
```

OUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
   -- Serial Number 0100219
   "Restricted Rights Legend"

- Use, duplication, or disclosure is subject to

- restrictions as set forth in subdivision (b) (3) (ii)

- of the rights in Technical Data and Computer

- Software Clause of FAR 52.227-7013. Manufacturer:

- Wizard software, 2171 S. Parfet Court, Lakewood,

- Colorado 80227 (1-303-987-1874)
   with Storage_Manager_Sequential;
  package body
Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator is
               type Node is
                                      The_Item : Item;
Next : Structure;
                           end record:
               procedure Free (The_Node : in out Node) is
               begin
              null;
end Free;
             Structure) is
             begin
                           The Node Next := To Next:
               end Set_Next;
               function Next Of (The Node : in Node) return Structure is
              return The_Node.Next;
end Next_Of;
             begin
                        in
Node_Manager.Free(To_The_Queue.The_Front);
To_The_Queue.The_Back := null;
if From_The_Queue.The_Front /= null then
    To_The_Queue.The_Front := Node_Manager.New_Item;
    To_The_Queue.The_Back := To_The_Queue.The_Front;
    To_The_Queue.The_Front.The_Item := From_Index.The_Item;
    To_Index := To_The_Queue.The_Front;
    From_Index := From_Index.Next;
    while From_Index /= null loop
    To_Index.Next := Node_Manager.New_Item;
    To_Index.Next := Node_Manager.New_Item;
    To_Index.Next := To_Index.Next.The_Item := From_Index.The_Item;
    To_Index.The_Item := From_Index.The_Item;
    To_Index.The_Item := To_Index.The_Item;
    To_Index.The_Item.The_Item;
    To_Index.The_Item.The_Item;
    To_Index.The_Item.The_Item.The_Item;
    To_Index.The_Item.The_Item.The_Item.The_Item;
    To_Index.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.The_Item.T
                                                  To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
                                     end loop:
                         end if;
             exception
when Storage_Error =>
                                     raise Overflow;
             end Copy;
            procedure Clear (The_Queue : in out Queue) is begin _
                        Node_Manager.Free(The_Queue.The_Front);
The_Queue.The_Back := null;
             end Clear:
           begin
if To_The_Queue.The_Front = null then
To_The_Queue.The_Front := Node_Manager.New_Item;
To_The_Queue.The_Front.The_Item := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Front;
Index := Index.Next;
                                     end loop;
if Previous = null then
    To_The_Queue.The_Front := Node_Manager.New_Item;
    To_The_Queue.The_Front.The_Item := The_Item;
```

```
end if;
elsif Index = null then
To_The_Queue.The_Back.Next := Node_Manager.New_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
To_The_Queue.The_Back.The_Item := The_Item;
                    Previous.Next := Node_Manager.New_Item;
                    Previous.Next.The_Item := The_Item;
Previous.Next.Next := Index;
        end if;
   exception
when Storage_Error =>
raise Overflow;
   end Add:
  procedure Pop (The_Queue : in out Queue) is
   Temporary_Node : Structure;
       exception
  when Constraint_Error =>
    raise Underflow;
end Pop;
  Positive) is
  begin
       un
while Index /= null loop
   if Count = At_The_Position then
     exit;
   else
                   Count := Count + 1:
       Count := Count + 1;
Previous := Index.;
Index := Index.Next;
end if;
end loop;
if Index = null then
raise Position_Error;
elsif Previous = null then
             From_The_Queue.The_Front := Index.Next;
        Previous.Next := Index.Next;
end if;
       end if;
if From_The_Queue.The_Back = Index then
    From_The_Queue.The_Back := Previous;
end if;
Index.Next := null;
  Node_Manager.Free(Index);
end Remove_Item;
modified by Tuan Nguyen replacing functions with procedures
  procedure Is_Equal
                                  Right
                                                       in Queue;
                                                     : out Boolean) is
       Result := Is_Equal(Left,Right);
  end Is_Equal;
                                (The_Queue
Result
                                                    : in Queue;
: out Natural) is
  procedure Length_Of
  begin
  Result := Length_Of(The_Queue);
end Length_Of;
  procedure Is_Empty
                                 (The_Queue
                                                     : in Oueue:
                                                     : out Boolean) is
  begin
       Result := Is_Empty(The_Queue);
  end Is_Empty;
  procedure Front_Of
                                                    : in Queue;
: Item) is
                                  Result
  begin
   Result := Front_Of(The_Queue);
end Front_Of;
 procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural) is
       Result := Position_Of(The_Item,In_The_Queue);
```

end Position_Of;

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
{\tt TYPE\ Queue\_Priority\_Balking\_Sequential\_Unbounded\_Managed\_Noniterator\ SPECIFICATION}
   GENERIC
      Item : PRIVATE TYPE.
      Priority Of : FUNCTION[The_Item : Item, RETURN : Priority],
func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
OPERATOR Copy
SPECIFICATION
     From_The_Queue : Queue,
To_The_Queue : Queue
OUTPUT
     To_The_Queue : Queue EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Clear
SPECIFICATION
     INPUT
The_Queue : Queue
     OUTPUT
     The_Queue : Queue EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Add
   SPECIFICATION
INPUT
        The_Item : Item,
     The_Item : Item,
To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
     EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Pop
SPECIFICATION
INPUT
The_Queue : Queue
     OUTPUT
     The_Queue : Queue EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Remove_Item SPECIFICATION
    INPUT
From_The_Queue : Queue,
At_The_Position : Positive
     OUTPUT
From_The_Queue : Queue
EXCEPTIONS
```

```
Overflow, Underflow, Position_Error
   END
   OPERATOR Is_Equal SPECIFICATION
      INPUT
Left : Queue,
Right : Queue
      OTTO
      Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
   OPERATOR Length_Of SPECIFICATION
      INPUT
The_Queue : Queue
OUTPUT
      Result : Natural EXCEPTIONS
        Overflow, Underflow, Position_Error
   END
   OPERATOR Is_Empty
   SPECIFICATION
INPUT
        The_Queue : Queue
     OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Underflow, Position_Error
  OPERATOR Front_Of
SPECIFICATION
INPUT
  INPUT
The Queue : Queue,
Result : Item
EXCEPTIONS
Overflow, Underflow, Position_Error
END
  OPERATOR Position_Of SPECIFICATION
      INPUT
     INFUT
The_Item : Item,
In_The_Queue : Queue
OUTPUT
Result : Natural
     EXCEPTIONS
        Overflow, Underflow, Position_Error
IMPLEMENTATION ADA

Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator
```

OUEUE PRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
   type Item is private;
   type Priority is limited private;
   with function Priority_Of (The_Item : in Item)
Priority;
   with function "<=" (Left : in Priority of The Item : in Item)</pre>
with function "<=" (Left : in Priority;
Right : in Priority) return Boolean;
package Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator
is
           type Queue(The_Size : Positive) is limited private;

        procedure Copy
        (From_The_Queue | : in out Queue);
        Queue;

        procedure Clear
        (The_Queue | : in out Queue);
        : in out Queue);

        procedure Add
        (The_Item | : in out Queue);
        : in out Queue);

        procedure Pop
        (The_Queue | : in out Queue);
        : in out Queue);

           procedure Pop (The_Queue
       modified by Tuan Nguyen replacing functions with procedures
                                                                                                             in Queue;
in Queue;
          procedure Is_Equal
                                                                     (Left
                                                                       Right
                                                                    Result
(The_Queue
Result
(The_Queue
                                                                                                             out Boolean);
in Queue;
out Natural);
          procedure Length_Of
          procedure Is_Empty
```

: in Oueue:

```
Result
                                                                                                : out Boolean):
          procedure Front_Of
                                                               (The_Queue
Result
 -- end of modification
         function Is_Equal (Left : in Queue; function Length_Of (The_Queue : in Queue) return Boolean; function Is_Empty (The_Queue : in Queue) return Natural; function Front_Of (The_Queue : in Queue) return Boolean; function Front_Of (The_Queue : in Queue) return Item;
                   with procedure Process (The_Item : in Item;
Continue : out Boolean);
         procedure Iterate (Over_The_Queue : in Queue);
         Overflow : exception;
Underflow : exception;
private
private
   type Items is array(Positive range <>) of Item;
   type Queue(The_Size : Positive) is
      record
      The_Back : Natural := 0;
      The_Items : Items(1 .. The_Size);
   end record;
end Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator;
```

OUEUE PRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body
Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator is
      begin
            if From_The_Queue.The_Back > To_The_Queue.The_Size then
    raise Overflow;
elsif From_The_Queue.The_Back = 0 then
                   To_The_Queue.The_Back := 0;
                   To_The_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
   From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
To_The_Queue.The_Back := From_The_Queue.The_Back;
             end if:
      end Copy;
      procedure Clear (The_Queue : in out Queue) is
      The_Queue.The_Back := 0;
end Clear;
      begin
            if To_The_Queue.The_Back = 0 then
   To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
The Item:
                  To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
                  while (Index <= To_The_Queue.The_Back) and then
   (Priority_Of(The_Item) <=
        Priority_Of(To_The_Queue.The_Items(Index))) loop
   Index := Index + 1;</pre>
                  end loop;
if Index > To_The_Queue.The_Back then
                         To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
                         To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
                   else
                         To_The_Queue.The_Items
((Index + 1) .. (To_The_Queue.The_Back + 1)) :=
To_The_Queue.The_Items(Index ..
           To_The_Queue.The_Back);
      exception
when Constraint_Error =>
raise Overflow;
      end Add:
      procedure Pop (The_Queue : in out Queue) is
           in
if The_Queue.The_Back = 0 then
    raise Underflow;
elsif The_Queue.The_Back = 1 then
    The_Queue.The_Back := 0;
                  The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
The_Queue.The_Items(2 .. The_Queue.The_Back);
The_Queue.The_Back := The_Queue.The_Back - 1;
            end if:
```

```
modified by Tuan Nguyen replacing functions with procedures
     procedure Is Equal
                                  (Left
                                                    : in Oueue:
                                   Right
Result
                                                    : in Queue;
: out Boolean) is
     begin
     Result := Is_Equal(Left,Right);
end Is_Equal;
     procedure Length_Of (The_Queue
                                                    : in Queue;
: out Natural) is
     begin
          Result := Length_Of(The_Queue);
     end Length_Of;
                                                    : in Queue;
     procedure Is Empty
                                 (The Oueue
                                                    : out Boolean) is
                                  Result
     begin
          Result := Is_Empty(The_Queue);
     end Is_Empty;
                                                    : in Queue;
: Item) is
     procedure Front Of
                                 (The Oueue
          Result := Front Of(The Oueue);
     end Front_Of;
   end of modification
    function Is_Equal (Left : in Queue; Right : in Queue) return Boolean is
          if Left.The_Back /= Right.The_Back then
         if Left.The_Back /= Right.The_Back then
    return False;
else
    for Index in 1 .. Left.The_Back loop
        if Left.The_Items(Index) /= Right.The_Items(Index)
then
                    return False;
end if;
          end loop;
return True;
end if;
     end Is_Equal;
     function Length_Of (The_Queue : in Queue) return Natural is
    return The_Queue.The_Back; end Length_Of;
     function Is_Empty (The_Queue : in Queue) return Boolean is
    return (The_Queue.The_Back = 0);
end Is_Empty;
     function Front_Of (The_Queue : in Queue) return Item is
         if The_Queue.The_Back = 0 then
               raise Underflow;
         return The_Queue.The_Items(1);
end if;
     end Front_Of;
    procedure Iterate (Over_The_Queue : in Queue) is
          Continue : Boolean;
          for The_Iterator in 1 .. Over_The_Queue.The_Back loop
   Process(Over_The_Queue.The_Items(The_Iterator), Continue);
   exit when not Continue;
          end loop;
```

end Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator;

QUEUE PRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
TYPE Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator
TYPE Queue_Priority_NonDalking_Sequential_Bounded_Managed_Iterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Priority: PRIVATE_TYPE,
Priority_Of: FUNCTION[The_Item: Item, RETURN: Priority],
func_"<=" : FUNCTION[Left: Priority, Right: Priority, RETURN:
Boolean?
   OOLEAN]
OPERATOR COPY
SPECIFICATION
INFUT
From_The_Queue : Queue,
To_The_Queue : Queue
OUTPUT
TTE_The_Queue : Queue
      To_The_Queue : Queue EXCEPTIONS
   END
   OPERATOR Clear
SPECIFICATION
INPUT
          The_Queue : Queue
      OUTPUT
The_Queue : Queue
EXCEPTIONS
          Overflow, Underflow
   END
   OPERATOR Add SPECIFICATION
      PECLIFICATION
INPUT
The_Item : Item,
To_The_Queue : Queue
      To_The_Queue : Queue EXCEPTIONS
          Overflow, Underflow
   END
   OPERATOR Pop
SPECIFICATION
      INPUT
The_Queue : Queue
OUTPUT
          The_Queue : Queue
   Overflow, Underflow
      EXCEPTIONS
   OPERATOR Is_Equal
SPECIFICATION
INPUT
```

```
Left : Queue,
Right : Queue
OUTFUT
Result : Boolean
EXCEPTIONS
        Overflow, Underflow
   END
   OPERATOR Length_Of
   SPECIFICATION
INPUT
The_Queue : Queue
OUTPUT
     Result : Natural 
EXCEPTIONS
        Overflow, Underflow
   END
  OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Queue : Queue
OUTPUT
        Result : Boolean
     EXCEPTIONS
Overflow, Underflow
   END
   OPERATOR Front_Of
SPECIFICATION
INPUT
     The_Queue : Queue,
Result : Item
EXCEPTIONS
        Overflow, Underflow
   END
   OPERATOR Iterate
SPECIFICATION
     GENERIC
Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t : Boolean]]
INPUT
     Over_The_Queue : Queue EXCEPTIONS
        Overflow, Underflow
  END
IMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Bounded_Managed_IteratorEND
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
generic
        type Item is private;
type Priority is limited private;
with function Priority_Of (The_Item : in Item)
                                                               (Left
Right
                                                                                    : in Priority;
: in Priority) return Boolean;
package
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator is
        type Queue is limited private;

        procedure
        Copy
        (From_The_Queue | : in out Queue);
        Queue | : in out Queue);

        procedure
        Clear (The_Queue | : in out Queue);
        : in out Queue);

        procedure
        Add | The_Ltem | : in out Queue);
        To_The_Queue | : in out Queue);

        procedure
        Pop (The_Queue | : in out Queue);

-- modified by Tuan Nguyen
-- replacing functions with procedures
        procedure Is_Equal
                                                       (Left
                                                                                     : in Queue;
: in Queue;
                                                         Right
                                                      Result
(The_Queue
                                                                                     : out Boolean) :
                                                                                    : in Queue;
: out Natural);
       procedure Length_Of
                                                         Result
        procedure Is_Empty
                                                      (The_Queue
                                                                                    : in Oueue;
```

```
procedure Front_Of Result : out Boolean);

(The_Queue : in Queue;
Result : Item);

-- end of modification

function Is_Equal (Left : in Queue)
    Right : in Queue) return Boolean;
function Length_Of (The_Queue : in Queue) return Natural;
function Is_Empty (The_Queue : in Queue) return Natural;
function Front_Of (The_Queue : in Queue) return Natural;
Overflow : exception;
Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
    record
    The_Front : Structure;
    end record;
end
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
 -- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"

Use, duplication, or disclosure is subject to
restrictions as set forth in subdivision (b) (3) (ii)
of the rights in Technical Data and Computer
Software Clause of FRR 52.227-7013. Manufacturer:
Wizard software, 2171 S. Parfet Court, Lakewood,
Colorado 80227 (1-303-987-1874)
 with Storage Manager Seguential:
 package body
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
       type Node is
    record
    The_Item : Item;
                    Next
                                   : Structure;
              end record;
       procedure Free (The_Node : in out Node) is
       begin
       null;
end Free;
      procedure Set_Next (The_Node : in out Node;
To_Next : in Struc
                                                                       Structure) is
       The_Node.Next := To_Next;
end Set_Next;
       function Next_Of (The_Node : in Node) return Structure is
       begin
             return The_Node.Next;
       end Next_Of;
       package Node Manager is new Storage Manager Sequential
                                                              (Item => Node,
Pointer => Structure,
Free => Free,
Set_Pointer => Set_Next,
                                                             (Item
                                                              Pointer_Of => Next_Of);
      begin
             in
Node_Manager.Free(To_The_Queue.The_Front);
To_The_Queue.The_Back := null;
if From_The_Queue.The_Front /= null then
To_The_Queue.The_Front := Node_Manager.New_Item;
To_The_Queue.The_Back := To_The_Queue.The_Front;
To_The_Queue.The_Front.The_Item := From_Index.The_Item;
To_Index := To_The_Queue.The_Front;
From_Index := From_Index.Next;
while_From_Index.The_Item := While_Front
                    while From_Index /= null loop

To_Index.Next := Node_Manager.New_Item;
To_Index.Next.The_Item := From_Index.The_Item;
To_Index := To_Index.Next;
                          From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
             end loop;
end if;
       exception
      when Storage_Error =>
    raise Overflow;
end Copy;
       procedure Clear (The_Queue : in out Queue) is
begin
             Node_Manager.Free(The_Queue.The_Front);
             The Oueue. The Back := null:
       procedure Add (The Item
             begin
             in
if To_The_Queue.The_Front = null then
   To_The_Queue.The_Front := Node_Manager.New_Item;
   To_The_Queue.The_Front.The_Item := The_Item;
   To_The_Queue.The_Back := To_The_Queue.The_Front;
To_The_Queue.The_Front := Node_Manager.New_Item;
```

```
To_The_Queue.The_Front.The_Item := The_Item;
To_The_Queue.The_Front.Next := Index;
if To_The_Queue.The_Back = null then
    To_The_Queue.The_Back := To_The_Queue.The_Front;
end if;
             elsif Index = null then
                   To_The_Queue.The_Back.Next := Node_Manager.New_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
To_The_Queue.The_Back.The_Item := The_Item;
                   Previous.Next := Node_Manager.New_Item;
Previous.Next.The_Item := The_Item;
Previous.Next.Next := Index;
      end if;
end if;
exception
when Storage_Error =>
raise Overflow;
end Add;
procedure Pop (The_Queue : in out Queue) is
   Temporary_Node : Structure;
      Temporary_Node := The_Queue.The_Front;
The_Queue.The_Front := The_Queue.The_Front.Next;
Temporary_Node.Next := null;
      .cumporary_Node.Next := null;
Node_Manager.Free(Temporary_Node);
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
end if;
exception
      when Constraint_Error =>
raise Underflow;
end Pop;
modified by Tuan Nguyen replacing functions with procedures
procedure Is_Equal
                                   (Left
                                                         : in Queue;
                                    Right
Result
                                                           in Oueue
      Result := Is_Equal(Left,Right);
end Is_Equal;
procedure Length_Of
                                 (The_Queue
                                                         : in Queue;
                                                         : out Natural) is
                                    Result
Result := Length_Of(The_Queue);
end Length_Of;
                                   (The_Queue
Result
                                                         : in Queue;
: out Boolean) is
procedure Is_Empty
      Result := Is_Empty(The_Queue);
end Is_Empty;
procedure Front_Of
                                                         : in Queue;
: Item) is
                                   (The_Queue
Result := Front_Of(The_Queue);
end Front_Of;
end of modification
begin
      while Left_Index /= null loop
if Left_Index.The_Item /= Right_Index.The_Item then
return False;
            else
Left_Index := Left_Index.Next;
      Right_Index := Right_Index.Next;
end if;
end loop;
return (Right_Index = null);
exception when Constraint_Error =>
            return False:
end Is_Equal;
function Length_Of (The_Queue : in Queue) return Natural is
      Count : Natural := 0;
Index : Structure := The_Queue.The_Front;
begin
while Index /= null loop
Count := Count + 1;
Index := Index.Next;
return Count;
end Length_Of;
function Is_Empty (The_Queue : in Queue) return Boolean is
begin
```

return (The_Queue.The_Front = null);

end Is_Empty;

function Front_Of (The_Queue : in Queue) return Item is
begin
 return The_Queue.The_Front.The_Item;
exception

when Constraint_Error =>
 raise Underflow;
end Front_Of;

end Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
TYPE
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
GENERIC
Item : PRIVATE_TYPE,
Priority : PRIVATE_TYPE,
Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
func_*<=': FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean|
Booleanl
    OPERATOR CODY
    OPERATOR COPY
SPECIFICATION
INPUT
From The Queue : Queue,
To The Queue : Queue
OUTPUT
To The Queue : Queue
EXCEPTIONS
             Overflow, Underflow
    END
    OPERATOR Clear
SPECIFICATION
        INPUT
The_Queue : Queue
OUTPUT
The_Queue : Queue
EXCEPTIONS
             Overflow, Underflow
    OPERATOR Add
    OPERATOR AND
SPECIFICATION
INPUT
The_Item : Item,
To_The_Queue : Queue
         OUTPUT
To_The_Queue : Queue
EXCEPTIONS
             Overflow, Underflow
    EMD
    OPERATOR Pop
SPECIFICATION
INPUT
         The_Queue : Queue
         The_Queue : Queue EXCEPTIONS
```

```
Overflow, Underflow
  OPERATOR IS_Equal
SPECIFICATION
INPUT
Left : Queue,
Right : Queue
OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Underflow
  OPERATOR Length_Of
SPECIFICATION
INPUT
        The_Queue : Queue
     OUTPUT
Result : Natural
EXCEPTIONS
         Overflow, Underflow
   OPERATOR Is_Empty
   SPECIFICATION
INPUT
The_Queue : Queue
OUTPUT
      Result : Boolean
EXCEPTIONS
  Overflow, Underflow
  OPERATOR Front_Of SPECIFICATION
     INPUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
         Overflow, Underflow
   END
END
IMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
END
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
generic
type Item is private;
package
package
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator is
      type Queue is limited private;
                                           (From_The_Queue
                                                                       : in
                                                                                     Oueue:
      procedure Copy
                                           To_The_Queue
(The_Queue
(The_Item
                                                                         in out Queue);
in out Queue);
in Item;
      procedure Clear
procedure Add
                                           (The_Queue : in out Queue);
(The_Queue : in out Queue);
(From_The_Queue : in out Queue;
At_The_Position : in Fositive);
     procedure Pop (The_Queue procedure Remove_Item (From_The_Queue
-- modified by Tuan Nguyen
-- replacing functions with procedures
                                                                    in Queue;
in Queue;
out Boolean);
in Queue;
out Natural);
in Queue;
      procedure Is_Equal
                                            Right
                                           Result
(The_Queue
Result
     procedure Length_Of
                                           (The_Queue
Result
(The_Queue
Result
      procedure Is_Empty
                                                                    out Boolean);
in Queue;
      procedure Front Of
                                                                    Item):
                                          (The_Item : in Item;
In_The_Queue : in Queue;
      procedure Position_Of
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
 package body
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
       type Node is
             record
                   The_Item : Item;
Next : Structure;
             end record;
      To_Index : Structure;
      begin
if From_The_Queue.The_Front = null then
To_The_Queue.The_Front := null;
To_The_Queue.The_Back := null;
else

To_The_Queue.The_Front :=
    new Node'(The_Item => From_Index.The_Item,
    Next => null);

To_The_Queue.The_Back := To_The_Queue.The_Front;

To_Index := To_The_Queue.The_Front;

From_Index := From_Index.Next;
while From_Index /= null loop
    To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,

Next => 2022110
                                                                                   => null);
                                                                    Next
                         To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
       end loop;
end if;
exception
when Storage_Error =>
                  raise Overflow;
      end Copy:
       procedure Clear (The_Queue : in out Queue) is
            The_Queue := Queue'(The_Front => null
                                             The_Back
       end Clear:
      procedure Add (The_Item : in Item;
To_The_Queue : in out Queue) is
      begin
            if To_The_Queue.The_Front = null then
                   To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                  Next =>:
To_The_Queue.The_Back := To_The_Queue.The_Front;
             else
                   To_The_Queue.The_Back.Next := new Node'(The_Item =>
            Next => r
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
end if;
       exception when Storage_Error =>
                  raise Overflow:
       end Add:
       procedure Pop (The_Queue : in out Queue) is
            in
The_Queue.The_Front := The_Queue.The_Front.Next;
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
end if;
       exception
when Constraint_Error =>
raise Underflow;
       end Pop;
      begin
             while Index /= null loop
                   if Count = At_The_Position then exit;
                   else
                         Count := Count + 1:
```

```
Previous := Index;
  Index := Index.Next;
end if;
          end loop;
if Index = null then
          raise Position_Error;
elsif Previous = null then
               From_The_Queue.The_Front := Index.Next;
          Previous.Next := Index.Next;
end if;
          if From_The_Queue.The_Back = Index then
    From_The_Queue.The_Back := Previous;
end if;
     end Remove_Item;
-- modified by Tuan Nguyen
-- replacing functions with procedures
     procedure Is Equal
                                                    : in Queue;
                                                     : in Oueue:
                                   Result
                                                     · out Boolean) is
     Result := Is_Equal(Left,Right);
end Is_Equal;
     procedure Length_Of (The_Queue
                                                     : out Natural) is
     Result := Length_Of(The_Queue);
end Length_Of;
     procedure Is Empty
                                  (The Oueue
                                                     : in Oueue:
                                   Result
                                                     : out Boolean) is
    Result := Is_Empty(The_Queue);
end Is_Empty;
     procedure Front_Of
                                  (The_Queue Result
     begin
          Result := Front_Of(The_Queue);
     end Front_Of;
    procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural) is
          Result := Position_Of(The_Item,In_The_Queue);
     end Position_Of;
-- end of modification
    begin
while Left_Index /= null loop
    if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
               else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
end if;
          end loop;
return (Right_Index = null);
     exception
          when Constraint_Error =>
     return False;
end Is_Equal;
     function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
     begin
          while Index /= null loop
          Count := Count + 1;
Index := Index.Next;
end loop;
          return Count:
     function Is_Empty (The_Queue : in Queue) return Boolean is
           eturn (The_Queue.The_Front = null);
     end Is_Empty;
     function Front_Of (The_Queue : in Queue) return Item is
     begin
return The Queue.The Front.The Item;
     exception
   when Constraint_Error =>
     raise Underflow;
end Front_Of;
          rosition_Of (The_Item : in Item;
In_The_Queue : in Queue) return Natural is
Position : Natural := 1;
     function Position_Of (The_Item
```

```
Index : Structure := In_The_Queue.The_Front;
begin
while Index /= null loop
    if Index.The_Item = The_Item then
        return Position;
    else
        Position := Position + 1;
        Index := Index.Next;
        end if;
    end loop;
    return 0;
end Position_Of;
```

```
procedure Iterate (Over_The_Queue : in Queue) is
    The_Iterator : Structure := Over_The_Queue.The_Front;
    Continue : Boolean;
begin
    while not (The_Iterator = null) loop
        Process(The_Iterator.The_Item, Continue);
        exit when not Continue;
        The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;
end Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
TYPE Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
     Item : PRIVATE_TYPE
  OPERATOR COPY
SPECIFICATION
INPUT
From The Queue : Queue,
      To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
      EXCEPTIONS
         Overflow, Underflow, Position_Error
  OPERATOR Clear SPECIFICATION
      INPUT
         The_Queue : Queue
     OUTPUT
The_Queue : Queue
EXCEPTIONS
         Overflow, Underflow, Position_Error
  OPERATOR Add
   OPERATOR Add
SPECIFICATION
INPUT
The_Item : Item,
To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
EXCEPPIONS
         Overflow, Underflow, Position_Error
   CINT
  OPERATOR Pop
SPECIFICATION
      INPUT
The_Queue : Queue
OUTPUT
         The Oueue : Oueue
      EXCEPTIONS
         Overflow, Underflow, Position_Error
  OPERATOR Remove_Item
SPECIFICATION
INPUT
        NPUT
From_The_Queue : Queue,
At_The_Position : Positive
      At_The_Position : Position
OUTPUT
From_The_Queue : Queue
EXCEPTIONS
         Overflow, Underflow, Position_Error
  OPERATOR Is_Equal
SPECIFICATION
INPUT
Left : Queue,
Right : Queue
```

```
OUTPUT
Result : Boolean
EXCEPTIONS
       Overflow, Underflow, Position_Error
  OPERATOR Length_Of
SPECIFICATION
INPUT
       The_Queue : Queue
    Result : Natural
EXCEPTIONS
       Overflow, Underflow, Position_Error
 OPERATOR IS_Empty
SPECIFICATION
INPUT
      The Oueue : Oueue
    Result : Boolean
EXCEPTIONS
       Overflow, Underflow, Position_Error
  OPERATOR Front_Of
  SPECIFICATION
    INPUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
 Overflow, Underflow, Position_Error
  OPERATOR Position Of
 OPERATOR POSITION_OF
SPECIFICATION
INPUT
The_Item : Item,
In_The_Queue : Queue
    OUTPRITT
    Result : Natural
EXCEPTIONS
       Overflow, Underflow, Position_Error
  END
  OPERATOR Iterate
  SPECIFICATION
    GENERIC
       Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
    INPUT
    Over_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
  END
IMPLEMENTATION ADA
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
```

OUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
generic
         type Item is private;
package
Paurage
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
is
        type Queue is limited private;

        procedure
        Copy
        (From_The_Queue : in out Queue);
        count Queue : in out Queue);

        procedure
        Clear (The_Queue : in out Queue);
        : in out Queue);

        procedure
        Add (The_Queue : in out Queue);
        : in out Queue);

        procedure
        Pop (The_Queue : in out Queue);

       modified by Tuan Nguyen replacing functions with procedures
                                                                                           : in Queue;
: in Queue;
: out Boolean);
: in Queue;
: out Natural);
: in Queue;
                                                           (Left
Right
Result
        procedure Is_Equal
                                                           (The_Queue
Result
(The_Queue
        procedure Length_Of
        procedure Is_Empty
                                                                                           : out Boolean);
: in Queue;
: Item);
                                                             Result
                                                           (The_Queue
Result
        procedure Front_Of
```

OUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                  "Restricted Rights Legend"
-- Wes, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of PAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
is
     type Node is
          record
                The_Item : Item;
                             : Structure:
           end record;
     To_Index

begin

if From_The_Queue.The_Front = null then

To_The_Queue.The_Front := null;

To_The_Queue.The_Back := null;
         From_Index.The_Item,
                                                                        => null):
                      To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
     end loop;
end if;
exception
when Storage_Error =>
                raise Overflow;
     procedure Clear (The_Queue : in out Queue) is begin
          The_Queue := Queue'(The_Front => null,
The_Back => null);
     end Clear;
     in
if To_The_Queue.The_Front = null then
To_The_Queue.The_Front := new Node'(The_Item => The_Next => null
                                                                               => null);
                To_The_Queue.The_Back := To_The_Queue.The_Front;
           else To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
          Next =>
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
end if;
                                                                                     => null);
     exception
           when Storage Error =>
     raise Overflow;
end Add;
     procedure Pop (The_Queue : in out Queue) is begin
          The_Queue.The_Front := The_Queue.The_Front.Next;
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
           end if;
     exception when Constraint_Error =>
     raise Underflow;
end Pop;
```

```
modified by Tuan Nguyen replacing functions with procedures
procedure Is_Equal
                             (Left
                                               : in Oueue:
                              Right
Result
begin
      Result := Is_Equal(Left, Right);
end Is_Equal;
procedure Length_Of (The_Queue
                                               : in Queue;
: out Natural) is
begin
      Result := Length_Of(The_Queue);
end Length_Of;
procedure Is Empty
                             (The Oueue
                                               : in Oueue;
                              Result
                                                : out Boolean) is
     Result := Is_Empty(The_Queue);
end Is_Empty;
procedure Front Of
                                               : in Queue;
: Item) is
                               Result
Result := Front_Of(The_Queue);
end Front_Of;
end of modification
begin
     while Left_Index /= null loop
if Left_Index.The_Item /= Right_Index.The_Item then
return False;
           Left_Index := Left_Index.Next;
   Right_Index := Right_Index.Next;
end if;
     end loop;
return (Right_Index = null);
exception when Constraint_Error =>
           return False:
end Is_Equal;
function Length_Of (The_Queue : in Queue) return Natural is
     Count : Natural := 0;
Index : Structure := The_Queue.The_Front;
begin
while Index /= null loop
Count := Count + 1;
Index := Index.Next;
end loop;
end Length_Of;
 function Is_Empty (The_Queue : in Queue) return Boolean is
      return (The_Queue.The_Front = null);
function Front Of (The Oueue : in Oueue) return Item is
      return The_Queue.The_Front.The_Item;
exception when Constraint_Error =>
raise Underflow;
end Front_Of;
procedure Iterate (Over_The_Queue : in Queue) is
   The_Iterator : Structure := Over_The_Queue.The_Front;
   Continue : Boolean;
begin
     while not (The_Iterator = null) loop
Process(The_Iterator.The_Item, Continue);
exit when not Continue;
The_Iterator := The_Iterator.Next;
      end loop:
end Iterate
```

Oueue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator;

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
TYPE
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
THE
TYPE
       INPUT
           From The Queue : Queue,
To The Queue : Queue
       OUTPUT
       To_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow
    END
   OPERATOR Clear
SPECIFICATION
       INPUT
The_Queue : Queue
OUTPUT
       The_Queue : Queue
EXCEPTIONS
Overflow, Underflow
    END
   OPERATOR Add
SPECIFICATION
       INPUT
The Item : Item,
       To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
       EXCEPTIONS
Overflow, Underflow
    END
    OPERATOR POP
SPECIFICATION
       INPUT
The_Queue : Queue
OUTPUT
The_Queue : Queue
EXCEPTIONS
           Overflow, Underflow
   OPERATOR Is_Equal
SPECIFICATION
INPUT
Left : Queue,
```

```
Right : Queue
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Underflow
   END
  OPERATOR Length_Of SPECIFICATION
    INPUT
The_Queue : Queue
     OUTPUT
Result : Natural
EXCEPTIONS
  Overflow, Underflow
  OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Queue : Queue
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Underflow
  OPERATOR Front_Of SPECIFICATION
     INPUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
  Overflow, Underflow
OPERATOR Iterate
SPECIFICATION
Process: PROCEDURE[The_Item : in[t : Item], Continue : out[t : Boolean]]
INFUT
       Over_The_Queue : Queue
  Overflow, Underflow
IMPLEMENTATION ADA
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch -- All Rights Reserved
 -- Serial Number 0100219
*Restricted Rights Legend*

- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator is
      type Node is
                 The_Item : Item;
Next : Structure;
            end record:
     in
if From_The_Queue.The_Front = null then
    To_The_Queue.The_Front := null;
    To_The_Queue.The_Back := null;
To_The_Queue.The_Back := null;
else

To_The_Queue.The_Front :=
    new Node' (The_Item => From_Index.The_Item,
    Next => null);
To_The_Queue.The_Back := To_The_Queue.The_Front;
To_Index := To_The_Queue.The_Front;
From_Index := From_Index.Next;
while From_Index /= null loop
    To_Index.Next := new Node' (The_Item =>
From_Index.The_Item,
    Next => null)
                                                              Next
                                                                           => null):
                       To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
           end loop;
end if;
      exception
    when Storage_Error =>
        raise Overflow;
end Copy;
      procedure Clear (The_Queue : in out Queue) is
      begin

The_Queue := Queue'(The_Front => null,

The_Back => null);
      end Clear;
     begin
           To_The_Queue.The_Back := To_The_Queue.The_Front;
else
                 To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item
                                                                                Next
                                                                                            =>
null);
                       To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
                 end if;
end if;
      exception when Storage_Error =>
      raise Overflow;
end Add;
      procedure Pop (The_Queue : in out Queue) is
```

```
begin
          The_Queue.The_Front := The_Queue.The_Front.Next;
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
          end if;
     exception
          when Constraint Error =>
     raise Underflow;
end Pop;
     Positive) is
          Previous : Structure;
Index : Structure := From_The_Queue.The_Front;
     begin
          while Index /= null loop
if Count = At_The_Position then
exit;
                else
                     Count := Count + 1;
                     Previous := Index;
Index := Index.Next;
                end if:
          end loop;
if Index = null then
               raise Position Error:
          elsif Previous = null then
From_The_Queue.The_Front := Index.Next;
          Previous.Next := Index.Next;
end if;
    end if;
if From_The_Queue.The_Back = Index then
    From_The_Queue.The_Back := Previous;
end if;
end Remove_Item;
-- modified by Tuan Nguyen
-- replacing functions with procedures
                                                     : in Queue;
: in Queue;
: out Boolean) is
     procedure Is_Equal
                                  (Left
                                   Right
Result
     begin
     Result := Is_Equal(Left,Right);
end Is_Equal;
     procedure Length_Of (The_Queue
                                                     : in Oueue;
                                                     · out Natural) is
          Result := Length_Of(The_Queue);
     end Length_Of;
    procedure Is_Empty
                                                     : out Boolean) is
                                   Result
    begin
   Result := Is_Empty(The_Queue);
end Is_Empty;
                                  (The_Queue
Result
     procedure Front_Of
                                                     : in Oueue;
          Result := Front_Of(The_Queue);
     end Front_Of;
    procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural) is
          Result := Position_Of(The_Item,In_The_Queue);
     end Position Of:
   end of modification
    while Left_Index /= null loop
    if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
                else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
end if;
          end loop;
return (Right_Index = null);
     exception when Constraint_Error =>
    return False;
end Is_Equal;
     function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
     begin
          while Index /= null loop
```

```
while Index /= null loop
    if Index.The_Item = The_Item then
        return Position;
    else
        Position := Position + 1;
        Index := Index.Next;
    end if;
    end loop;
    return 0;
end Position_Of;

procedure Iterate (Over_The_Queue : in Queue) is
        The_Iterator : Structure := Over_The_Queue.The_Front;
        Continue : Boolean;
begin
    while not (The_Iterator = null) loop
        Process(The_Iterator.The_Item, Continue);
        exit when not Continue;
        The_Iterator := The_Iterator.Next;
    end loop;
end Iterate;
end Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
TYPE Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator SPECIFICATION
   GENERIC
      Trem · PRIVATE TYPE
      ltem: FRIVATE_TYPE,
Priority: PRIVATE_TYPE,
Priority_Of: FUNCTION[The_Item : Item, RETURN : Priority],
func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :</pre>
Boolean]
OPERATOR Copy
SPECIFICATION
      INPUT
From_The_Queue : Queue,
To_The_Queue : Queue
      To_The_Queue : Queue EXCEPTIONS
         Overflow, Underflow, Position_Error
   OPERATOR Clear
   SPECIFICATION
      INPUT
The_Queue : Queue
      OUTPUT
The Queue : Queue
EXCEPTIONS
         Overflow, Underflow, Position_Error
  OPERATOR Add SPECIFICATION
      INPUT
         NPUT
The_Item : Item,
To_The_Queue : Queue
     To_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
   END
  OPERATOR Pop
SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
The_Queue : Queue
EXCEPTIONS
         Overflow, Underflow, Position_Error
  OPERATOR Remove_Item
  OFERATOR Remove_Item
SPECIFICATION
INPUT
From_The_Queue : Queue,
At_The_Position : Positive
OUTPUT
From_The_Queue : Queue
EXCEPTIONS
EXCEPTIONS
EXCEPTION Underflow Position
         Overflow, Underflow, Position_Error
  OPERATOR Is_Equal
  SPECIFICATION
INPUT
```

```
Left : Queue,
Right : Queue
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Length_Of
SPECIFICATION
INPUT
        The_Queue : Queue
     OUTPUT
     Result : Natural
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR IS_Empty
SPECIFICATION
     The_Queue : Queue
     Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Front_Of SPECIFICATION
    INPUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Position_Of
  SPECIFICATION
INPUT
The_Item : Item,
     In_The_Queue : Queue
OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Iterate SPECIFICATION
     GENERIC
        Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean11
     INPUT
     Over_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
  END
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
generic
        type Item is private;
type Priority is limited private;
with function Priority_Of (The_Item : in Item)
Priority;
with function "<="
                                                                    (Left
Right
                                                                                         : in Priority;
: in Priority) return Boolean;
package
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator is
         type Queue is limited private;

        procedure
        Copy
        (From_The_Queue | : in out Queue);
        Queue;

        procedure
        Clear (The_Queue | : in out Queue);
        : in out Queue);

        procedure
        Add (The_Item | : in out Queue);

        To_The_Queue | : in out Queue);
        : in out Queue);

        procedure
        Pop (The_Queue | : in out Queue);

       modified by Tuan Nguyen replacing functions with procedures
                                                                                              in Queue;
in Queue;
out Boolean);
in Queue;
                                                           (Left
        procedure Is_Equal
                                                           Right
Result
(The_Queue
Result
        procedure Length Of
                                                                                               out Natural);
                                                           (The_Queue
Result
                                                                                           : in Queue;
: out Boolean);
        procedure Is_Empty
```

```
procedure Front_Of (The_Queue : in Queue;
Result : Item);

-- end of modification

function Is_Equal (Left : in Queue;
Right : in Queue) return Boolean;
function Length_Of (The_Queue : in Queue) return Natural;
function Is_Empty (The_Queue : in Queue) return Boolean;
function Front_Of (The_Queue : in Queue) return Item;

generic
with procedure Process (The_Item : in Item;
Continue : out Boolean);
procedure Iterate (Over_The_Queue : in Queue);

Overflow : exception;
Underflow : exception;
Underflow : exception;
private
type Node;
type Structure is access Node;
type Queue is
record
The_Front : Structure;
The_Back : Structure;
end record;
end Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
      type Node is
            record
                  The_Item : Item;
Next : Structure;
            end record;
      in
if From_The_Queue.The_Front = null then
   To_The_Queue.The_Front := null;
   To_The_Queue.The_Back := null;
To_The_Queue.The_Back := null;
else

To_The_Queue.The_Front :=
    new Node'(The_Item => From_Index.The_Item,
    Next => null);

To_The_Queue.The_Back := To_The_Queue.The_Front;
    To_Index := To_The_Queue.The_Front;
    From_Index := From_Index.Next;
    while From_Index /= null loop

To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
    Next => null);
                        To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
            end loop;
end if;
      exception when Storage_Error =>
                 raise Overflow;
      end Copy;
      procedure Clear (The_Queue : in out Queue) is
      begin The_Queue := Queue'(The_Front => null, The_Back => null);
      To_The_Queue.The_Back := To_The_Queue.The_Front;
                  while (Index /= null) and then
(Priority_Of(The_Item) <=
Priority_Of(Index.The_Item) | loop
Previous := Index;
Index := Index.Next;
                  To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
null):
                        To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
                  else
                       end if;
end if;
      exception
when Storage_Error =>
raise Overflow;
      end Add:
```

```
procedure Pop (The_Queue : in out Queue) is begin
          exception
when Constraint_Error =>
raise Underflow;
end Pop;
    modified by Tuan Nguyen replacing functions with procedures
                                 (Left
                                                   : in Queue;
: in Queue;
    procedure Is Equal
                                  Right
                                                   : out Boolean) is
    begin
          Result := Is_Equal(Left,Right);
     end Is_Equal;
                                                   : in Oueue:
    procedure Length_Of (The_Queue
                                                    : out Natural) is
    Result := Length_Of(The_Queue);
end Length_Of;
    procedure Is_Empty
                                 (The_Queue
Result
                                                   : in Oueue:
                                                   : out Boolean) is
    begin
    Result := Is_Empty(The_Queue);
end Is_Empty;
    procedure Front_Of
                                 (The_Queue
Result
                                                   : in Oueue:
          Result := Front_Of(The_Queue);
    end Front_Of;
    begin
         while Left_Index /= null loop
   if Left_Index.The_Item /= Right_Index.The_Item then
      return False;
               Left_Index := Left_Index.Next;
   Right_Index := Right_Index.Next;
end if;
         end loop;
return (Right_Index = null);
    exception
          when Constraint_Error =>
    return False;
end Is_Equal;
    function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
          while Index /= null loop
              Count := Count + 1;
Index := Index.Next;
         end loop;
return Count;
    end Length_Of;
     function Is_Empty (The_Queue : in Queue) return Boolean is
          return (The_Queue.The_Front = null);
     function Front_Of (The_Queue : in Queue) return Item is
    begin return The_Queue.The_Front.The_Item;
    exception when Constraint_Error =>
    raise Underflow;
end Front_Of;
    procedure Iterate (Over_The_Queue : in Queue) is
   The_Iterator : Structure := Over_The_Queue.The_Front;
   Continue : Boolean;
    begin
         while not (The_Iterator = null) loop
Process(The_Iterator.The_Item, Continue);
exit when not Continue;
The_Iterator := The_Iterator.Next;
          end loop;
     end Iterate;
end Oueue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
TYPE Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Priority: PRIVATE_TYPE,
Priority_Of: FUNCTION[The_Item: Item, RETURN: Priority],
func_"<=":FUNCTION[Left: Priority, Right: Priority, RETURN: Boolean]
     OPERATOR Copy
SPECIFICATION
          INPUT
         INPUT
From_The_Queue : Queue,
To_The_Queue : Queue
OUTFUT
              To_The_Queue : Queue
         EXCEPTIONS
Overflow, Underflow
     END
     OPERATOR Clear
SPECIFICATION
    SPECIFICATION
INFUT
The_Queue: Queue
OUTFUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END
    OPERATOR Add
SPECIFICATION
INPUT
The_Item : Item,
To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
EXCEPTIONS
               Overflow, Underflow
      END
     OPERATOR POD
      SPECIFICATION
INPUT
          The_Queue : Queue
OUTPUT
          The_Queue : Queue
EXCEPTIONS
Overflow, Underflow
      END
     OPERATOR IS_Equal SPECIFICATION
          INPUT
```

```
Left : Queue,
Right : Queue
      OUTPUT
     Result : Boolean
EXCEPTIONS
Overflow, Underflow
   END
  OPERATOR Length_Of SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
      Result : Natural
EXCEPTIONS
Overflow, Underflow
   ENTO
  OPERATOR Is_Empty
SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
        Result : Boolean
      EXCEPTIONS
        Overflow, Underflow
   END
  OPERATOR Front_Of
SPECIFICATION
INPUT
      The Queue : Queue,
Result : Item
EXCEPTIONS
        Overflow, Underflow
   END
   OPERATOR Iterate
SPECIFICATION
      GENERIC
Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t : Boolean]]
      INPUT
      Over_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow
   END
IMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
generic
type Item is private;
package
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator is
       type Queue is limited private;
       procedure Copy (From_The_Queue : in out Queue;
procedure Clear (The_Queue : in out Queue);
procedure Add (The_Item : in out Queue);
procedure Pop (The_Queue : in out Queue);
procedure Pop (The_Queue : in out Queue);
procedure Remove_Item (From_The_Queue : in out Queue);
At_The_Position : in out Queue);
      modified by Tuan Nguyen replacing functions with procedures
                                                                               : in Queue;
: in Queue;
: out Boolean);
       procedure Is_Equal
                                                    (Left
                                                     Result
                                                                               : in Queue;
: out Natural);
: in Queue;
                                                    (The_Queue
Result
(The_Queue
       procedure Length_Of
       procedure Is_Empty
                                                                               : out Boolean);
: in Queue;
: Item);
                                                      Result
                                                    (The_Queue
       procedure Front_Of
                                                     Result
       procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
```

OUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator is
        type Node is
               record
                       The_Item : Item;
Next : Structure;
                Next
end record;
        procedure Free (The_Node : in out Node) is
begin
        null;
end Free;
       procedure Set_Next (The_Node : in out Node; To_Next : in Struc
                                                                                     Structure) is
        begin
        The_Node.Next := To_Next;
end Set_Next;
        function Next_Of (The_Node : in Node) return Structure is
        return The_Node.Next;
end Next_Of;
       begin
                Node Manager.Free(To The Oueue.The Front);
               Node_Manager.Free(To_The_Queue.The_Front);
To_The_Queue.The_Back := null;
To_The_Queue.The_Front /= null then
    To_The_Queue.The_Front /= Node_Manager.New_Item;
    To_The_Queue.The_Front := Node_Manager.New_Item;
    To_The_Queue.The_Back := To_The_Queue.The_Front;
    To_Index := To_The_Queue.The_Front;
    From_Index := To_The_Queue.The_Front;
    From_Index /= null loop
    To_Index.Next := Node_Manager.New_Item;
    To_Index.Next.The_Item := From_Index.The_Item;
    To_Index := To_Index.Next;
    From_Index := To_Index.Next;
    From_Index := To_Index.Next;
    To_The_Queue.The_Back := To_Index;
end loop;
                end loop;
end if;
        exception
        when Storage_Error =>
    raise Overflow;
end Copy;
        procedure Clear (The_Queue : in out Queue) is begin
                In
Node_Manager.Free(The_Queue.The_Front);
The_Queue.The_Back := null;
         end Clear:
        in
if To_The_Queue.The_Front = null then
To_The_Queue.The_Front := Node_Manager.New_Item;
To_The_Queue.The_Front.The_Item := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Front;
                 else
   To_The_Queue.The_Back.Next := Node_Manager.New_Item;
   To_The_Queue.The_Back.Next.The_Item:= The_Item;
   To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
                 end if;
        exception
when Storage_Error =>
raise Overflow;
        procedure Pop (The_Queue : in out Queue) is
   Temporary_Node : Structure;
```

```
begin
           Temporary_Node := The_Queue.The_Front;
The_Queue.The_Front := The_Queue.The_Front.Next;
Temporary_Node.Next := null;
Node_Manager.Free(Temporary_Node);
          if The_Queue.The_Front = null then
The_Queue.The_Back := null;
end if;
     exception
when Constraint_Error =>
raise Underflow;
     end Pop;
     Trevious : Structure;
Index : Structure := From_The_Queue.The_Front;
     begin
           while Index /= null loop
if Count = At_The_Position then
exit;
                else
                Count := Count + 1;
Previous := Index;
Index := Index.Next;
end if;
           end loop;
if Index = null then
    raise Position_Error;
elsif Previous = null then
                From_The_Queue.The_Front := Index.Next;
                Previous.Next := Index.Next;
          Previous.Next := Index.Next;
end if;
if From_The_Queue.The_Back = Index then
From_The_Queue.The_Back := Previous;
end if;
           Index.Next := null:
     Node_Manager.Free(Index);
end Remove_Item;
-- modified by Tuan Nguyen
-- replacing functions with procedures
                                                       : in Queue;
     procedure Is_Equal
                                     Right
                                     Result
                                                        : out Boolean) is
     Result := Is_Equal(Left, Right);
end Is_Equal;
                                                       : in Queue;
: out Natural) is
     procedure Length_Of
                                    (The Oueue
     begin
           Result := Length_Of(The_Queue);
     end Length_Of;
                                                       : in Queue;
: out Boolean) is
     procedure Is_Empty
                                    (The_Queue
                                     Result
     begin
  Result := Is_Empty(The_Queue);
end Is_Empty;
     procedure Front_Of
                                    (The_Queue
                                                        : in Queue;
           Result := Front_Of(The_Queue);
     end Front_Of;
     procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natur
                                                       : out Natural) is
     Result := Position_Of(The_Item,In_The_Queue);
end Position_Of;
     in
while Left_Index /= null loop
    if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
    else
                      Left Index := Left Index.Next;
                 Right_Index := Right_Index.Next;
end if;
           end loop;
return (Right_Index = null);
      exception
when Constraint_Error =>
                return False;
      end Is_Equal;
```

```
function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
begin
   while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
   end loop;
   return Count;
end Length_Of;
function Is_Empty (The_Queue : in Queue) return Boolean is
begin
   return (The_Queue.The_Front = null);
end Is_Empty;
function Front_Of (The_Queue : in Queue) return Item is
begin
   return The_Queue.The_Front.The_Item;
exception
   when Constraint_Error =>
        raise Underflow;
end Front_Of;
function Position_Of (The_Item : in Item;
        In_The_Queue : in Queue) return Natural is
```

```
Position: Natural := 1;
    Index : Structure := In_The_Queue.The_Front;

begin
    while Index /= null loop
        if Index.The_Item = The_Item then
            return Position;
        else
            Position := Position + 1;
            Index := Index.Next;
        end if;
        end loop;
        return 0;
end Position_Of;

procedure Iterate (Over_The_Queue : in Queue) is
        The_Iterator : Structure := Over_The_Queue.The_Front;
        Continue : Boolean;

begin
    while not (The_Iterator = null) loop
        Frocess(The_Iterator.The_Item, Continue);
        exit when not Continue;
        The_Iterator := The_Iterator.Next;
    end loop;
end Iterate;
end Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
GENERIC
Item : PRIVATE_TYPE
OPERATOR COpy
SPECIFICATION
INPUT
From The Queue : Cucus
         From_The_Queue : Queue,
      To_The_Queue : Queue
OUTPUT
To_The_Queue : Queue
EXCEPTIONS
          Overflow, Underflow, Position_Error
   OPERATOR Clear
   SPECIFICATION
INPUT
         The_Queue : Queue
       OUTPUT
         Overflow, Underflow, Position_Error
    END
   OPERATOR Add
   OPERATOR Add
SPECIFICATION
INPUT
The_Item : Item,
To_The_Queue : Queue
      OUTPUT
To_The_Queue : Queue
EXCEPTIONS
          Overflow, Underflow, Position_Error
    END
   OPERATOR Pop
SPECIFICATION
      INPUT
The_Queue : Queue
OUTPUT
          The_Queue : Queue
      EXCEPTIONS
Overflow, Underflow, Position_Error
   OPERATOR Remove_Item SPECIFICATION
       INPUT
         From_The_Queue : Queue,
At_The_Position : Positive
          From_The_Queue : Queue
      EXCEPTIONS
Overflow, Underflow, Position_Error
    OPERATOR IS_Equal
SPECIFICATION
INPUT
Left : Queue,
```

Right : Queue

```
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Length_Of
  SPECIFICATION
INPUT
     The_Queue : Queue
OUTPUT
     Result : Natural EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
  OPERATOR Is_Empty
SPECIFICATION
     TINPUT
The_Queue : Queue
OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Underflow, Position_Error
  OPERATOR Front_Of SPECIFICATION
     INPUT
     The_Queue : Queue,
Result : Item
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Position_Of SPECIFICATION
     INPUT
The_Item : Item,
In_The_Queue : Queue
     Result : Natural
EXCEPTIONS
Overflow, Underflow, Position_Exror
   מוגים
OPERATOR Iterate
SPECIFICATION
     GENERIC
Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
     INPUT
     Over_The_Queue : Queue EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator
END
IMPLEMENTATION ADA
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
   Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
     type Node is
           record
           The_Item : Item;
Next : Structure;
end record;
     begin

if From_The_Queue.The_Front = null then

To_The_Queue.The_Front := null;

To_The_Queue.The_Back := null;
To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
                 end loop:
           end if;
      exception
when Storage_Error =>
                raise Overflow;
      end Copy;
      procedure Clear (The_Queue : in out Queue) is
           The_Queue := Queue'(The_Front => null
     if To_The_Queue.The_Front = null then
   To_The_Queue.The_Front := new Node (The_Item => The_Item,
                                                                   Next
                 To_The_Queue.The_Back := To_The_Queue.The_Front;
           else
                 To_The_Queue.The_Back.Next := new Node'(The_Item =>
           Next => r
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
end if;
     end 11,
exception
when Storage_Error =>
raise Overflow;
     procedure Pop (The_Queue : in out Queue) is begin
           The Onene The Front := The Onene.The Front.Next:
           if The_Queue.The_Front := ine_Queue.If
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
end if;
      exception
when Constraint_Error =>
raise Underflow;
end Pop;
      Previous : Structure;
Index : Structure := From_The_Queue.The_Front;
      begin
           mwhile Index /= null loop
  if Count = At_The_Position then
     exit;
```

```
else
                      Count := Count + 1;
Previous := Index;
Index := Index.Next;
                 end if:
           end 1r;
end loop;
if Index = null then
    raise Position_Error;
elsif Previous = null then
                 From_The_Queue.The_Front := Index.Next;
                 Previous.Next := Index.Next;
           Previous.Neac
end if;
if From_The_Queue.The_Back = Index then
    From_The_Queue.The_Back := Previous;
     end if;
end Remove_Item;
    modified by Tuan Nguyen replacing functions with procedures
                                     (Left
                                                         : in Queue;
     procedure Is_Equal
                                                         : in Queue;
: out Boolean) is
     Result := Is_Equal(Left,Right);
end Is_Equal;
                                                         : in Queue;
: out Natural) is
     procedure Length_Of {The_Queue Result
     Result := Length_Of(The_Queue);
end Length_Of;
     procedure Is_Empty
                                     (The_Queue
Result
                                                         : in Oueue:
                                                          : out Boolean) is
     begin
           Result := Is_Empty(The_Queue);
     end Is_Empty;
     procedure Front_Of
                                     (The_Queue
                                                         : in Queue;
: Item) is
     Result := Front_Of(The_Queue);
end Front_Of;
     procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural) is
     Result := Position_Of(The_Item,In_The_Queue);
end Position_Of;
-- end of modification
     begin
           while Left_Index /= null loop
if Left_Index.The_Item /= Right_Index.The_Item then
return False;
           else
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
end if;
end loop;
return (Right_Index = null);
      exception
when Constraint_Error =>
     return False;
end Is_Equal;
      function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
            while Index /= null loop
                Count := Count + 1;
Index := Index.Next;
           end loop;
return Count;
      function Is_Empty (The_Queue : in Queue) return Boolean is
      return (The_Queue.The_Front = null);
end Is_Empty;
      function Front_Of (The_Queue : in Queue) return Item is
      begin
           return The_Queue.The_Front.The_Item;
      exception
when Constraint_Error =>
raise Underflow;
end Front_Of;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
TYPE
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
TNDUM
TNDUM
       From_The_Queue : Queue,
To_The_Queue : Queue
OUTPUT
       To_The_Queue : Queue EXCEPTIONS
   Overflow, Underflow, Position_Error
   OPERATOR Clear
SPECIFICATION
       INPUT
          The_Queue : Queue
       OUTPUT
The_Queue : Queue
EXCEPTIONS
          Overflow, Underflow, Position_Error
   OPERATOR Add
   OPERATOR AGG
SPECIFICATION
INPUT
The_Item : Item,
To_The_Queue : Queue
       OUTPUT
To_The_Queue : Queue
EXCEPTIONS
          Overflow, Underflow, Position_Error
    END
   OPERATOR Pop
SPECIFICATION
       INPUT
The_Queue : Queue
OUTPUT
       OUTPUT
The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
   OPERATOR Remove_Item
SPECIFICATION
INPUT
      INPUT
From_The_Queue : Queue,
At_The_Position : Positive
OUTPUT
From_The_Queue : Queue
           Overflow, Underflow, Position_Error
```

```
OPERATOR Is_Equal SPECIFICATION
     INPUT
        Left : Queue,
Right : Queue
     ייווקייונס
     Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  EVID
  OPERATOR Length_Of SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
Result : Natural
     EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR IS_Empty
SPECIFICATION
INPUT
        The Oueue : Queue
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Front_Of SPECIFICATION
     INPUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Position_Of
  OPERATOR Position_Or
SPECIFICATION
INPUT
The_Item : Item,
In_The_Queue : Queue
OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Underflow, Position_Error
  END
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
generic
type Item is private;
package
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
    is
           type Queue is limited private;

        procedure
        Copy
        (From_The_Queue
        : in

        To_The_Queue
        : in

        procedure
        Clear
        (The_Queue
        : in

        procedure
        Add
        (The_Item
        : in

        To_The_Queue
        : in
        in

        procedure
        Pop
        (The_Queue
        : in

                                                                                                      : in Queue;

: in out Queue);

: in out Queue);

: in Item;

: in out Queue);

: in out Queue);
-- modified by Tuan Nguyen
-- replacing functions with procedures
                                                                                                                   : in Queue;
: in Queue;
: out Boolean);
: in Queue;
                                                                           (Left
           procedure Is_Equal
                                                                             Right
Result
          procedure Length_Of
                                                                          (The_Queue
Result
(The_Queue
                                                                                                                   : in Queue;
: out Natural);
: in Queue;
: out Boolean);
          procedure Is_Empty
```

Result

```
(The_Queue
Result
        procedure Front_Of
                                                                                     : Item);
-- end of modification
        function Is_Equal (Left : in Queue; Right : in Queue) return Boolean; function Length_Of (The_Queue : in Queue) return Natural; function Is_Empty (The_Queue : in Queue) return Boolean; function Front_Of (The_Queue : in Queue) return Item;
        Overflow : exception;
Underflow : exception;
private
        rate
type Node;
type Structure is access Node;
type Queue is
record
The_Front : Structure;
The_Back : Structure;
                end record:
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
      type Node is
           record
The_Item : Item;
                 Next
                             : Structure;
            end record;
     begin
           in
if From_The_Queue.The_Front = null then
   To_The_Queue.The_Front := null;
   To_The_Queue.The_Back := null;
                else
From_Index.The_Item.
                                                                          => null):
                       To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
                 end loop;
           end if;
      exception
      when Storage_Error =>
    raise Overflow;
end Copy;
     end Clear:
     procedure Add (The_Item : in Item;
To_The_Queue : in out Queue) is
           if To_The_Queue.The_Front = null then
   To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                Next =>:
To The Queue.The Back := To The Queue.The Front;
           else
                 To_The_Queue.The_Back.Next := new Node'(The_Item =>
The Item.
           To_The_Queue.The_Back := To_The_Queue.The_Back.Next; end if;
      exception when Storage_Error =>
      raise Overflow; end Add;
      procedure Pop (The_Queue : in out Queue) is
       egin
   The_Queue.The_Front := The_Queue.The_Front.Next;
if The_Queue.The_Front = null then
        The_Queue.The_Back := null;
```

```
exception
when Constraint_Error =>
              raise Underflow:
-- modified by Tuan Nguyen
-- replacing functions with procedures
    procedure Is Equal
                                 (Left
                                  Result
                                                   : out Boolean) is
    begin
          Result := Is_Equal(Left, Right);
     end Is_Equal;
                                                   : in Queue;
: out Natural) is
    procedure Length_Of (The_Queue
                                  Result
    Result := Length_Of(The_Queue);
end Length_Of;
                                 (The_Queue
Result
                                                   : in Queue;
: out Boolean) is
     procedure Is_Empty
    begin
          Result := Is_Empty(The_Queue);
     end Is_Empty;
    procedure Front Of
                                 (The_Queue
Result
                                                   : in Queue;
          Result := Front_Of(The_Queue);
     end Front_Of;
-- end of modification
    while Left_Index /= null loop
   if Left_Index.The_Item /= Right_Index.The_Item then
        return False;
               else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
end if;
          end loop;
return (Right_Index = null);
     exception
when Constraint_Error =>
               return False:
     end Is_Equal;
     function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
          while Index /= null loop
               Count := Count + 1;
Index := Index.Next;
     end loop;
return Count;
end Length_Of;
     function Is_Empty (The_Queue : in Queue) return Boolean is
     begin
  return (The_Queue.The_Front = null);
end Is_Empty;
     function Front_Of (The_Queue : in Queue) return Item is
         return The_Queue.The_Front.The_Item;
     exception
when Constraint_Error =>
raise Underflow;
end Front_Of;
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
TYPE
Queue_Nompriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
x
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
INPUT
From_The_Queue: Queue
OUTPUT
To_The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Clear
SPECIFICATION
INPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Add
SPECIFICATION
INPUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Add
SPECIFICATION
INPUT
The_Ltem: Item,
To_The_Queue: Queue
OUTPUT
O_The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
The_Queue: Queue
OUTPUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
OTPUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
OUTPUT
OPERATOR Pop
SPECIFICATION
INPUT
The_Queue: Queue
EXCEPTIONS
Overflow, Underflow
OUTPUT
OU
```

```
OPERATOR Is_Equal SPECIFICATION
     TINFUT
Left: Queue,
Right: Queue
OUTPUT
Result: Boolean
EXCEPTIONS
         Overflow, Underflow
  END
  OPERATOR Length_Of
SPECIFICATION
INPUT
The_Queue : Queue
OUTPUT
     Result : Natural
EXCEPTIONS
Overflow, Underflow
  END
  OPERATOR IS_Empty
SPECIFICATION
     INPUT
The_Queue : Queue
OUTPUT
      Result : Boolean
EXCEPTIONS
         Overflow, Underflow
  END
  OPERATOR Front Of
  OPERATOR Front_Or
SPECIFICATION
INFUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
Overflow, Underflow
  END
IMPLEMENTATION ADA
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
r
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
rric
type Item is private;
type Priority is limited private;
with function Priority_Of (The_Item : in Item)
Priority;
with function "<="
                                                                                      : in Priority;
Right : in Friority;

Right : in Friority; return Boolean;
package Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
is
        type Queue is limited private;
                                                      (From_The_Queue : in Queue;
To_The_Queue : in out Queue);
(The_Queue : in out Queue);
(The_Item : in Item;
                                                       (From_The_Queue : in To_The_Queue : in (The_Queue : in The_Queue : in To_The_Queue : in To_The_Queue : in At_The_Position : in
       procedure Clear
procedure Add
                                                                                            : in out Oueue):
       procedure Pop
procedure Remove_Item
                                                                                               in out Queue;
in out Queue;
                                                                                                               Positive):
       modified by Tuan Nguyen replacing functions with procedures
                                                                                    : in Queue;
: in Queue;
: out Boolean);
: in Queue;
: out Natural);
: in Queue;
: out Boolean);
: in Queue;
: Item);
       procedure Is Equal
                                                       (Left
                                                      (Left
Right
Result
(The_Queue
Result
(The_Queue
Result
(The_Queue
Result
       procedure Length Of
       procedure Is_Empty
       procedure Front_Of
```

```
procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural);

-- end of modification

function Is_Equal (Left : in Queue;
Right : in Queue) return Boolean;
function Length_Of (The_Queue : in Queue) return Natural;
function Is_Empty (The_Queue : in Queue) return Natural;
function Front_Of (The_Queue : in Queue) return Natural;
function Position_Of (The_Item : in Item;
In_The_Queue : in Queue) return Natural;

generic
with procedure Process (The_Item : in Item;
Continue : out Boolean);
procedure Iterate (Over_The_Queue : in Queue);

Overflow : exception;
Underflow : exception;
Position_Error : exception;

private
type Node;
type Structure is access Node;
type Queue is
record
The_Front : Structure;
The_Back : Structure;
end record;
end Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (c) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body
Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator is
        type Node is
                e Node 1s
record
The_Item : Item;
Next : Structure;
end record;
        procedure Free (The_Node : in out Node) is begin _
                 mull:
         end Free;
        procedure Set_Next (The_Node : in out Node;
To_Next : in Struc
                                                                                          Structure) is
                 The_Node.Next := To_Next;
         end Set_Next;
         function Next_Of (The_Node : in Node) return Structure is
                 return The Node.Next;
          end Next_Of;
        package Node_Manager is new Storage_Manager_Sequential (Item => Node, Pointer => Structu
                                                                               To_Index : Structure,
in Node Manager.Free(To_The_Queue.The_Front);
To_The_Queue.The_Back := null;
if From_The_Queue.The_Front /= null then
    To_The_Queue.The_Front := Node_Manager.New_Item;
    To_The_Queue.The_Back := To_The_Queue.The_Front;
    To_The_Queue.The_Front.The_Item := From_Index.The_Item;
    To_Index := To_The_Queue.The_Front;
    From_Index := From_Index.Next;
    while From_Index /= null loop
    To_Index.Next := Node_Manager.New_Item;
    To_Index.Next.The_Item := From_Index.The_Item;
    To_Index := To_Index.Next;
    From_Index := To_Index.Next;
    From_Index := To_Index.Next;
    To_The_Queue.The_Back := To_Index;
end loop;
         begin
                 ro_Th
end loop;
end if;
         exception
when Storage_Error =>
raise Overflow;
         end Copy;
         procedure Clear (The_Queue : in out Queue) is
                  n
Node_Manager.Free(The_Queue.The_Front);
The_Queue.The_Back := null;
         begin
if To_The_Queue.The_Front = null then
To_The_Queue.The_Front := Node_Manager.New_Item;
To_The_Queue.The_Front.The_Item := The_Item;
To_The_Queue.The_Back := To_The_Queue.The_Front;
                  while (Index /= null) and then

(The Item) <-
 while (Index /= null) and the (Priority_Of(The_Item)
Priority_Of(Index.The_Item) loop
Previous := Index;
Index := Index.Next;
                          Index := Almost
end loop;
if Previous = null then
   To_The_Queue.The_Front := Node_Manager.New_Item;
   To_The_Queue.The_Front.The_Item := The_Item;
```

```
To_The_Queue.The_Front.Next := Index;
if To_The_Queue.The_Back = null then
    To_The_Queue.The_Back := To_The_Queue.The_Front;
               end if;
elsif Index = null then
                      In Index = null then
To_The_Queue.The_Back.Next := Node_Manager.New_Item;
To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
To_The_Queue.The_Back.The_Item := The_Item;
                      Previous.Next := Node_Manager.New_Item;
Previous.Next.The_Item := The_Item;
Previous.Next.Next := Index;
       end if;
exception
when Storage_Error =>
raise Overflow;
end Add;
procedure Pop (The_Queue : in out Queue) is
   Temporary_Node : Structure;
      Temporary_Node := The_Queue.The_Front;
The_Queue.The_Front := The_Queue.The_Front.Next;
Temporary_Node.Next := null;
Node_Manager.Free(Temporary_Node);
if The_Queue.The_Front = null then
The_Queue.The_Back := null;
end if;
begin
 exception
when Constraint_Error =>
raise Underflow;
 end Pop;
procedure Remove_Item (From_The_Queue : in out Queue;
At_The_Position : in Positi

Count : Natural := 1;
Previous : Structure;
Index : Structure := From_The_Queue.The_Front;
                                                                                          Positive) is
begin
       un
while Index /= null loop
   if Count = At_The_Position then
        exit;
   else
                      Count := Count + 1;
               Previous := Index;
Index := Index.Next;
end if;
       end 1:;
end loop;
if Index = null then
    raise Position_Error;
elsif Previous = null then
    From_The_Queue.The_Front := Index.Next;
        Previous.Next := Index.Next; end if;
        if From_The_Queue.The_Back = Index then
    From_The_Queue.The_Back := Previous;
end if;
         Index.Next := null;
 Node_Manager.Free(Index);
end Remove_Item;
modified by Tuan Nguyen replacing functions with procedures
                                                                    : in Queue;
 procedure Is_Equal
                                            Right
         Result := Is_Equal(Left, Right);
 end Is_Equal;
                                          (The_Queue
Result
                                                                    : in Queue;
: out Natural) is
 procedure Length_Of
 begin
 Result := Length_Of(The_Queue);
end Length_Of;
                                                                     : in Queue;
: out Boolean) is
                                          (The_Queue
Result
 procedure Is_Empty
         Result := Is_Empty(The_Queue);
 end Is Empty:
                                                                     : in Queue;
: Item) is
 procedure Front_Of
                                           (The Oueue
 begin
         Result := Front_Of(The_Queue);
  end Front_Of;
 procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural) is
         Result := Position_Of(The_Item,In_The_Queue);
```

end Position_Of;

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator SPECIFICATION GENERIC
      Item : PRIVATE TYPE,
Item : PRIVATE_TYPE,
Priority : PRIVATE_TYPE,
Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
OPERATOR Copy
SPECIFICATION</pre>
       INPUT
         From_The_Queue : Queue
To_The_Queue : Queue
      OUTPUT
      To_The_Queue : Queue EXCEPTIONS
         Overflow, Underflow, Position_Error
   OPERATOR Clear SPECIFICATION
      INPUT
          The_Queue : Queue
      OUTPUT
      The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
   END
   OPERATOR Add
SPECIFICATION
      INPUT
The_Item : Item,
To_The_Queue : Queue
      OUTPUT
      To_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
   END
   OPERATOR Pop
SPECIFICATION
      INPUT
      The_Queue : Queue
          The_Queue : Queue
      EXCEPTIONS
Overflow, Underflow, Position_Error
   OPERATOR Remove_Item SPECIFICATION
      INPUT
         From_The_Queue : Queue,
At_The_Position : Positive
      From_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
   ENID
   OPERATOR Is_Equal SPECIFICATION
```

INPUT

```
Left : Queue
     Right : Queue
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Underflow, Position_Error
  OPERATOR Length_Of SPECIFICATION
     INPUT
     The_Queue : Queue
       Result : Natural
     EXCEPTIONS
Overflow, Underflow, Position_Error
  END
  OPERATOR IS_Empty
SPECIFICATION
        The_Queue : Queue
     OUTPUT
Result : Boolean
EXCEPTIONS
       Overflow, Underflow, Position_Error
  OPERATOR Front_Of SPECIFICATION
     INPUT
The Queue : Queue,
Result : Item
EXCEPTIONS
Overflow, Underflow, Position_Error
  END
  OPERATOR Position_Of SPECIFICATION
     The Item : Item,
In The Queue : Queue
OUTPUT
Result : Natural
EXCEPTIONS
OVERFILE Underflow.
        Overflow, Underflow, Position_Error
  END
  OPERATOR Iterate SPECIFICATION
     GENERIC
        Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
     INPUT
     Over_The_Queue : Queue EXCEPTIONS
       Overflow, Underflow, Position_Error
  END
Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
generic
       tric
type Item is private;
type Priority is limited private;
with function Priority_Of (The_Item : in Item)
Priority;
with function "<="
                                                                               : in Priority;
: in Priority) return Boolean;
package
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator is
        type Queue is limited private;
       procedure Copy

To_The_Queue

procedure Clear
procedure Add

To_The_Queue

To_The_Queue

To_The_Queue

Procedure Pop
procedure Remove_Item

The Ducue

The Queue

From_The_Queue

The_Queue

The_Queue

The_Queue
                                                                                    : in Queue;
: in out Queue);
: in out Queue);
: in Item;
                                                                                     : in out Queue);
: in out Queue);
: in out Queue;
                                                     At_The_Position : in
                                                                                                       Positive):
      modified by Tuan Nguyen replacing functions with procedures
                                                                               : in Queue;
: in Queue;
: out Boolean);
: in Queue;
: out Natural);
: in Queue;
       procedure Is_Equal
                                                   (Left
                                                   Right
Result
(The_Queue
Result
       procedure Length_Of
       procedure Is_Empty
                                                   (The_Queue
```

```
Result
(The_Oueue
                                                                                : out Boolean);
        procedure Front_Of
                                                                                : in Queue;
: Item);
                                                      Result
       procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural);
 -- end of modification
                                                                             : in Queue;
: in Queue) return Boolean;
: in Queue) return Natural;
: in Queue) return Boolean;
                                                   (Left
        function Is_Equal
        function Is_Equal (Left
Right
function Length_Of (The_Queue
function Is_Empty (The_Queue
function Front_Of (The_Queue
function Position_Of (The_Item
In_The_Queue
                                                                              : in Queue) return Item;
: in Item;
                                                                             : in Queue) return Natural;
        Overflow : exception;
Underflow : exception;
Position_Error : exception;
private
        type Node;
type Structure is access Node;
type Queue is
record
record
The_Front : Structure;
The_Back : Structure;
end record;
end Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
is
     type Node is
          record
The_Item : Item;
               Next
                          : Structure:
          end record;
     To_Index : Structure;
     begin
         in
if From_The_Queue.The_Front = null then
   To_The_Queue.The_Front := null;
   To_The_Queue.The_Back := null;
else
                                                                  => null);
                    To_Index := To_Index.Next;
From_Index := From_Index.Next;
To_The_Queue.The_Back := To_Index;
         To_The end loop; end if;
     exception
when Storage_Error =>
raise Overflow;
     end Copy:
    procedure Clear (The_Queue : in out Queue) is begin
         The_Queue := Queue'(The_Front => null
                                   The_Back => null);
    if To_The_Queue.The_Front = null then
To_The_Queue.The_Front := new Node'(The_Item => The_It
Next => null);
               To_The_Queue.The_Back := To_The_Queue.The_Front;
               while (Index /= null) and then
(Priority_Of(The_Item)
Priority_Of(Index.The_Item)) loop
Previous := Index;
Index := Index.Next;
               end loop;
if Previous = null then
              The Item.
                                                                       Next
mul1):
                    To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
               else
                    end if;
end if;
     exception
when Storage_Error =>
raise Overflow;
     end Add:
```

```
procedure Pop (The_Queue : in out Queue) is begin
          inthe_Queue.The_Front := The_Queue.The_Front.Next;
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
           end if;
     exception
           when Constraint_Error =>
     raise Underflow;
end Pop;
     Positive) is
                      : Structure := From_The_Queue.The_Front;
           Index
     begin
          while Index /= null loop
                if Count = At_The_Position then
    exit;
                else
                     Count := Count + 1;
Previous := Index;
Index := Index.Next;
          end if;
end loop;
if Index = null then
          raise Position_Error;
elsif Previous = null then
From_The_Queue.The_Front := Index.Next;
          else
Previous.Next := Index.Next;
          if From_The_Queue.The_Back = Index then
    From_The_Queue.The_Back := Previous;
          end if:
     end Remove_Item;
-- modified by Tuan Nguyen
-- replacing functions with procedures
     procedure Is_Equal
                                                     : in Oueue;
                                    Right
                                                     : out Boolean) is
          Result := Is_Equal(Left,Right);
     end Is_Equal;
     procedure Length_Of (The_Queue Result
                                                     : in Queue;
: out Natural) is
     begin
          Result := Length_Of(The_Queue);
     end Length_Of;
                                  (The Oueue
                                                     : in Oueue;
     procedure Is_Empty
                                                     : out Boolean) is
          Result := Is_Empty(The_Queue);
     end Is_Empty;
                                                     : in Queue;
: Item) is
     procedure Front_Of
                                   Result
     Result := Front_Of(The_Queue);
end Front_Of;
     procedure Position_Of (The_Item : in Item;
In_The_Queue : in Queue;
Result : out Natural) is
     Result := Position_Of(The_Item,In_The_Queue);
end Position_Of;
-- end of modification
     begin
          while Left_Index /= null loop
if Left_Index.The_Item /= Right_Index.The_Item then
return False;
                else
   Left_Index := Left_Index.Next;
   Right_Index := Right_Index.Next;
end if;
     end loop;
return (Right_Index = null);
exception
when Constraint_Error =>
               return False:
     end Is_Equal;
     function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
```

OUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
{\tt TYPE\ Queue\_Priority\_Balking\_Sequential\_Unbounded\_Unmanaged\_Noniterator\ SPECIFICATION}
      ENERIC
Item : PRIVATE_TYPE,
Priority : PRIVATE_TYPE,
Priority : PRIVATE_TYPE,
Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN : lean]</pre>
   GENERIC
Boolean)
OPERATOR Copy
SPECIFICATION
       INPIR
          From_The_Queue : Queue,
To_The_Queue : Queue
       OUTPUT
      To_The_Queue : Queue EXCEPTIONS
          Overflow, Underflow, Position_Error
   OPERATOR Clear
   OPERATOR CLEAR
SPECIFICATION
INPUT
The_Queue : Queue
          The_Queue : Queue
       EXCEPTIONS
          Overflow, Underflow, Position_Error
   OPERATOR Add
SPECIFICATION
      INPUT
The_Item : Item,
To_The_Queue : Queue
OUTPUT
      OUTPUT
To_The_Queue : Queue
EXCEPTIONS
Overflow, Underflow, Position_Error
   OPERATOR Pop
SPECIFICATION
      INPUT
      INPUT
The_Queue : Queue
OUTPUT
The_Queue : Queue
EXCEPTIONS
          Overflow, Underflow, Position_Error
   OPERATOR Remove_Item
   SPECIFICATION
INPUT
      INPUT
From_The_Queue : Queue,
At_The_Position : Positive
OUTPUT
From_The_Queue : Queue
EXCEPTIONS
```

```
Overflow, Underflow, Position_Error
   END
   OPERATOR Is_Equal SPECIFICATION
      TMPITT
      Left : Queue,
Right : Queue
OUTPUT
      Result : Boolean
EXCEPTIONS
Overflow, Underflow, Position_Error
   OPERATOR Length_Of SPECIFICATION
     PECIFICATION
INPUT
The_Queue : Queue
OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Underflow, Position_Error
   OPERATOR Is_Empty
   SPECIFICATION
      INPUT
        The_Queue : Queue
     OUTPUT
Result : Boolean
     EXCEPTIONS
         Overflow, Underflow, Position_Error
   OPERATOR Front_Of
SPECIFICATION
INPUT
        The_Queue : Queue,
      Result : Item
EXCEPTIONS
        Overflow, Underflow, Position_Error
   END
   OPERATOR Position_Of SPECIFICATION
     TNPITT
     INPUT
The_Item : Item,
In_The_Queue : Queue
OUTPUT
Result : Natural
     EXCEPTIONS
Overflow, Underflow, Position_Error
   END
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
Result : out Natural);

procedure Is_Empty (The_Queue : in Queue;
    Result : out Boolean);

procedure Front_Of (The_Queue : in Queue;
    Result : in Queue;
    Result : in Queue;
    Result : Item);

-- end of modification

function Is_Equal (Left : in Queue) return Boolean;
function Length_Of (The_Queue : in Queue) return Natural;
function Is_Empty (The_Queue : in Queue) return Boolean;
function Front_Of (The_Queue : in Queue) return Boolean;
function Front_Of (The_Queue : in Queue) return Item;

Overflow : exception;
Underflow : exception;
private
    type Node;
    type Structure is access Node;
    type Structure is access Node;
    type Queue is
    record
    The_Front : Structure;
    The_Back : Structure;
    end record;
end

Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
 -- Serial Number 0100219
                     "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body
   Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
       type Node is
             record
                    The_Item : Item;
Next : Structure;
             end record;
      in
if From_The_Queue.The_Front = null then
    To_The_Queue.The_Front := null;
    To_The_Queue.The_Back := null;
To_The_Queue.The_Back := null;
else

To_The_Queue.The_Front :=
    new Node' (The_Item => From_Index.The_Item,
    Next => null);
To_The_Queue.The_Back := To_The_Queue.The_Front;
To_Index := To_The_Queue.The_Front;
From_Index := From_Index.Next;
while From_Index /= null loop
    To_Index.Next := new Node' (The_Item =>
From_Index.The_Item,
    Next => null);
                                                                                      => null);
                          To_Index := To_Index.Next;
From_Index := From_Index.Next;
                           To_The_Queue.The_Back := To_Index;
             end loop;
end if;
       exception
when Storage_Error =>
raise Overflow;
       end Copy;
       procedure Clear (The_Queue : in out Queue) is
             end Clear:
       procedure Add (The_Item : in Item;
To_The_Queue : in out Queue) is
             Previous : Structure;
Index : Structure := To_The_Queue.The_Front;
       begin
             in
if To_The_Queue.The_Front = null then
To_The_Queue.The_Front := new Node'(The_Item => The_
To_The_Queue.The_Front := new Node'(The_Item => The_Tront == null)
                                                                               Next
                                                                                              => null):
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                    while (Index /= null) and then
while (Index /= mult) and then
(Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
Previous := Index;
Index := Index.Next;
                    end loop;
if Previous = null then
                   To_The_Queue.The_Back.Next := new Node'(The_Item =>
The Item.
null);
                          To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
                    else
                          end if:
```

```
end if:
     exception when Storage_Error =>
    raise Overflow;
end Add;
     procedure Pop (The_Queue : in out Queue) is begin
         Int
The_Queue.The_Front := The_Queue.The_Front.Next;
if The_Queue.The_Front = null then
    The_Queue.The_Back := null;
    exception
   when Constraint_Error =>
      raise Underflow;
end Pop;
-- modified by Tuan Nguyen
-- replacing functions with procedures
    procedure Is_Equal
                                                   : in Queue;
                                                  : in Queue;
: out Boolean) is
                                  Right
                                  Result
         Result := Is_Equal(Left,Right);
     end Is_Equal;
     procedure Length_Of (The_Queue
                                  Result
                                                   : out Natural) is
          Result := Length_Of(The_Queue);
     end Length_Of;
                                                  : in Queue;
: out Boolean) is
                                (The_Queue
    procedure Is Empty
    Result := Is_Empty(The_Queue);
end Is_Empty;
     procedure Front Of
                                (The_Queue
                                                   : in Queue;
: Item) is
    begin
         Result := Front_Of(The_Queue);
     end Front_Of;
    while Left_Index /= null loop
    if Left_Index.The_Item /= Right_Index.The_Item then
                   return False:
                    Left_Index := Left_Index.Next;
                    Right_Index := Right_Index.Next;
         end if;
end loop;
          return (Right_Index = null);
     exception when Constraint_Error =>
               return False:
     end Is_Equal;
    function Length_Of (The_Queue : in Queue) return Natural is
   Count : Natural := 0;
   Index : Structure := The_Queue.The_Front;
         while Index /= null loop
Count := Count + 1;
Index := Index.Next;
         end loop;
     return Cou
end Length_Of;
     function Is_Empty (The_Queue : in Queue) return Boolean is
    begin return (The_Queue.The_Front = null);
end Is_Empty;
     function Front_Of (The_Queue : in Queue) return Item is
         return The_Queue.The_Front.The_Item;
     exception
when Constraint_Error =>
raise Underflow;
     end Front_Of;
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
TYPE
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
GENERIC
    Item: PRIVATE_TYPE,
        Priority: PRIVATE_TYPE,
        Priority: FIVATE_TYPE,
        Priority: FIVATE_TYPE,
        Priority_Of: FUNCTION[The_Item: Item, RETURN: Priority],
        func_*c=*: FUNCTION[Left: Priority, Right: Priority, RETURN:
Boolean]
    OPERATOR Copy
SPECIFICATION
    INPUT
        From_The_Queue: Queue
        COTHE_Queue: Queue
        EXCEPTIONS
        Overflow, Underflow
END

OPERATOR Clear
SPECIFICATION
    INPUT
        The_Queue: Queue
    EXCEPTIONS
        Overflow, Underflow
END

OPERATOR Add
SPECIFICATION
    INPUT
        The_Queue: Queue
        EXCEPTIONS
        Overflow, Underflow
END

OPERATOR Add
SPECIFICATION
    INPUT
    The_Item: Item,
        To_The_Queue: Queue
    EXCEPTIONS
        Overflow, Underflow
END

OPERATOR POP
SPECIFICATION
    INPUT
    The_Queue: Queue
    EXCEPTIONS
        Overflow, Underflow
END

OPERATOR POP
SPECIFICATION
    INPUT
    The_Queue: Queue
    EXCEPTIONS
        Overflow, Underflow
END

OPERATOR POP
SPECIFICATION
    INPUT
    The_Queue: Queue
    EXCEPTIONS
        Overflow, Underflow
END

OPERATOR POP
SPECIFICATION
    INPUT
    The_Queue: Queue
    EXCEPTIONS
        Overflow, Underflow
END
```

```
Overflow, Underflow
   END
   OPERATOR Is Equal
  OPERATOR Is Equal
SPECIFICATION
INPUT
Left: Queue,
Right: Queue
OUTPUT
Result: Boolean
EXCEPTIONS
          Overflow, Underflow
   END
  OPERATOR Length_Of
   SPECIFICATION
INPUT
         The_Queue : Queue
      OUTPUT
Result : Natural
EXCEPTIONS
         Overflow, Underflow
   END
  OPERATOR IS_Empty
SPECIFICATION
INPUT
     INFUT
The_Queue : Queue
OUTFUT
Result : Boolean
EXCEPTIONS
Overflow, Underflow
   END
  OPERATOR Front_Of
SPECIFICATION
INPUT
The_Queue : Queue,
Result : Item
EXCEPTIONS
         Overflow, Underflow
  END
HMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
END
```

RING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
generic
generic
    type Item is private;
package Ring_Sequential_Bounded_Managed_Iterator is
     type Ring(The_Size : Positive) is limited private;
     type Direction is (Forward, Backward);
                                 (From_The_Ring
                                                     · in
    procedure Copy
Ring;
                                 To_The_Ring
Ring);
procedure Clear
Ring);
                                 (The_Ring
                                                     : in out
procedure Insert
Item;
                                 (The_Item
                                                     : in
                                 In_The_Ring
                                                     : in out
Ring);
                                                     : in out
procedure Pop
Ring);
                                 (The Ring
procedure Rotate
                                 (The_Ring
                                                     : in out
                                  In_The_Direction : in
procedure Mark (The_Ring Ring);
procedure Rotate_To_Mark (The_Ring Ring);
                                                     : in out
                                                     : in out
    modified by Tuan Nguyen
10 January 1996
adding procedures to replace functions
```

```
(The_Ring : in Ring;
Result : out Item);
(The_Ring : in Ring;
Result : out Boolean);
     procedure Top_Of
     procedure At_Mark
-- end of modification
    function Is_Equal (Left : in Ring;
Right : in Ring) return
Boolean:
     function Extent_Of (The_Ring : in Ring) return
Natural;
function Is_Empty (The_Ring : in Ring) return
Boolean:
Boolean;
function Top_Of (The_Ring : in Ring) return
Item;
function At_Mark (The_Ring : in Ring) return
Boolean:
     generic
         with procedure Process (The_Item : in Item;
Continue : out
Boolean):
    procedure Iterate (Over_The_Ring : in Ring);
    Overflow
                     : exception;
    Underflow : exception;
Rotate_Error : exception;
    The_Back : Natural := 0;
The_Back : Natural := 0;
The_Items : Items(1 .. The_Size);
          end record;
end Ring_Sequential_Bounded_Managed_Iterator;
```

RING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
 Booch
-- All Rights Reserved
 -- Serial Number 0100219
                          "Restricted Rights Legend"
-- Vse, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)

    (ii)

            of the rights in Technical Data and Computer
            software Clause of FAR 52.227-7013. Manufacturer:
            Wizard software, 2171 S. Parfet Court, Lakewood,
            Colorado 80227 (1-303-987-1874)

package body Ring_Sequential_Bounded_Managed_Iterator is
        procedure Copy (From_The_Ring : in Ring;
    To_The_Ring : in out Ring) is
begin
   if From_The_Ring.The_Back >
To_The_Ring.The_Size then
   raise Overflow;
   elsif From_The_Ring.The_Back = 0 then
        To_The_Ring.The_Top := 0;
        To_The_Ring.The_Back := 0;
        To_The_Ring.The_Back := 0;
        To_The_Ring.The_Mark := 0;
}
        begin
else
    To_The_Ring.The_Items(1 ..
From_The_Ring.The_Back) :=
    From_The_Ring.The_Items(1 ..
From_The_Ring.The_Back);
                else
From_The_Ring.The_Back):
    To_The_Ring.The_Top :=
From_The_Ring.The_Back :=
To_The_Ring.The_Back: =
From_The_Ring.The_Back: =
To_The_Ring.The_Mark: =
From_The_Ring.The_Mark;
end if;
         end Copy;
        procedure Clear (The_Ring : in out Ring) is begin
                 n
The_Ring.The_Top := 0;
The_Ring.The_Back := 0;
The_Ring.The_Mark := 0;
         end Clear:
        begin
                 if In_The_Ring.The_Back = In_The_Ring.The_Size
                raise Overflow;
elsif In_The_Ring.The_Back = 0 then
   In_The_Ring.The_Top := 1;
   In_The_Ring.The_Back := 1;
   In_The_Ring.The_Mark := 1;
   In_The_Ring.The_Items(1) := The_Item;
else
else
In_The_Ring.The_Items
((In_The_Ring.The_Top + 1) ..
(In_The_Ring.The_Back + 1)) :=
In_The_Ring.The_Items(In_The_Ring.The_Top
In_The_Ring.The_Back);
                        In_The_Ring.The_Items(In_The_Ring.The_Top)
                        In_The_Ring.The_Back :=
end if;
end Insert;
        procedure Pop(The_Ring : in out Ring) is begin
          rocedure roptaling rocedure roptaling if The Ring.The_Back = 0 then raise Underflow; elsif The_Ring.The_Back = 1 then The_Ring.The_Top := 0; The_Ring.The_Back := 0; The_Ring.The_Mark := 0;
                        The_Ring.The_Items(The_Ring.The_Top ..
The_Ring.The_Back - 1)) :=
The_Ring.The_Items((The_Ring.The_Top + 1)
 THE_RING.INC_STAR.

The_Ring.The_Back);
The_Ring.The_Back := The_Ring.The_Back - 1;
if The_Ring.The_Top > The_Ring.The_Back
```

```
if The_Ring.The_Top = The_Ring.The_Mark
then
                           The Ring. The Mark := 1:
                     end if;
The_Ring.The_Top := 1;
                else
                     if The_Ring.The_Mark > The_Ring.The_Top
                           The_Ring.The_Mark :=
          end if;
end if;
end if;
epp;
The_Ring.The_Mark - 1;
     end Pop;
    : in out Ring;
Direction) is
     begin
          in
if The_Ring.The_Back = 0 then
    raise Rotate_Error;
elsif In_The_Direction = Forward then
    The_Ring.The_Top := The_Ring.The_Top + 1;
    if The_Ring.The_Top > The_Ring.The_Back
then
               The_Ring.The_Top := 1;
end if;
                The Ring.The Top := The Ring.The Top - 1;
if The Ring.The Top = 0 then
The Ring.The Top := The Ring.The Back;
end if;
     end if;
end Rotate;
      procedure Mark (The_Ring : in out Ring) is
     begin
The_Ring.The_Mark := The_Ring.The_Top;
    procedure Rotate_To_Mark (The_Ring : in out Ring)
     begin
     The_Ring.The_Top := The_Ring.The_Mark;
end Rotate_To_Mark;
    modified by Tuan Nguyen
10 January 1996
adding procedures to replace functions
     procedure Is_Equal (Left
                                               : in Ring:
                                  Right : in Ring;
Result : out Boolean) is
     begin
          Result := Is_Equal(Left,Right);
     end Is_Equal;
    begin
     Result := Extent_Of(The_Ring);
end Extent_Of;
     procedure Is_Empty (The_Ring : in Ring;
Result : out Boolean) is
           Result := Is_Empty(The_Ring);
     end Is_Empty;
                                (The_Ring : in Ring;
Result : out Item) is
     procedure Top_Of
     Result := Top_Of(The_Ring);
end Top_Of;
     procedure At_Mark (The_Ring : in Ring;
Result : out Boolean) is
     begin
           Result := At_Mark(The_Ring);
     end At_Mark;
-- end of modification
     Left_Index : Natural := Left.The_Top;
Right_Index : Natural := Right.The_Top;
     begin
if Left.The_Back /= Right.The_Back then
ir Lert.The_Back /= Right.The_Back then
    return False;
elsif Left.The_Items(Left_Index) /=
Right.The_Items(Right_Index) then
    return False;
elsif (Left.The_Mark = Left_Index) and then
    (Right.The_Mark /= Right_Index) then
    return False;
else
```

```
Left_Index := Left_Index + 1;
if Left_Index > Left.The_Back then
    Left_Index := 1;
end if;
Right_Index := Right_Index + 1;
if Right_Index > Right_The_Back then
    Right_Index := 1;
end if;
while Left_Index /= Left.The_Top loop
    if Left_Index /= Left_Index /=
                                    if Left_index /= Left_index) /=
   Right.The_Items(Left_Index) /=
   Right.The_Items(Right_Index) then
   return False;
elsif (Left.The_Mark = Left_Index) and
then
                                               (Right.The_Mark /= Right_Index)
                                              return False;
                                     else
                                              e
Left_Index := Left_Index + 1;
if Left_Index > Left.The_Back then
Left_Index := 1;
end if;
                                               end ir;
Right_Index := Right_Index + 1;
if Right_Index > Right.The_Back
then
                 Right_Index := 1;
end if;
end loop;
end loop;
return (Right_Index = Right.The_Top);
end if;
        exception
  when Constraint_Error =>
    return (Left.The_Top = Right.The_Top);
        end Is_Equal;
         function Extent_Of (The_Ring : in Ring) return
Natural is
         begin
        return The_Ring.The_Back;
end Extent_Of;
```

```
function Is Empty (The_Ring : in Ring) return
Boolean is
begin
     return (The_Ring.The_Back = 0);
end Is_Empty;
     function Top_Of (The_Ring : in Ring) return Item is
          return The_Ring.The_Items(The_Ring.The_Top);
     exception
          when Constraint_Error =>
     raise Underflow;
end Top_Of;
     function At_Mark (The_Ring : in Ring) return
Boolean is
     begin
     return (The_Ring.The_Top = The_Ring.The_Mark);
end At_Mark;
     procedure Iterate (Over_The_Ring : in Ring) is
   Continue : Boolean := True;
begin
          for The_Iterator in Over_The_Ring.The_Top ..
Over_The_Ring.The_Back loop
Process(Over_The_Ring.The_Items(The_Iterator),
Continue);
Continue);
exit when not Continue;
end loop;
if Continue then
for The_Iterator in 1 ..
Over_The_Ring.The_Top - 1 loop
    end loop;
end Iterate;
Process(Over_The_Ring.The_Items(The_Iterator),
Continue);
end Ring_Sequential_Bounded_Managed_Iterator;
```

RING SEQUENTIAL BOUNDED MANAGED ITERATOR

```
TYPE Ring_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_Ring: Ring,
To_The_Ring: Ring
OUTPUT
TO_The_Ring: Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
   OPERATOR Clear
SPECIFICATION
       TNPUT
      INPUT
The_Ring : Ring
OUTPUT
The_Ring : Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
   OPERATOR Insert
SPECIFICATION
       INPUT
      INPUT
The_Item : Item,
In_The_Ring : Ring
OUTPUT
In_The_Ring : Ring
       EXCEPTIONS
Overflow, Underflow, Rotate_Error
   OPERATOR FOD
SPECIFICATION
INPUT
The_Ring : Ring
OUTPUT
The_Ring : Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
   END
   OPERATOR Rotate
   OFERATOR ROLLES
SPECIFICATION
INPUT
The_Ring: Ring,
In_The_Direction: Direction
      OUTPUT
The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   CMS
   OPERATOR Mark
SPECIFICATION
      PECIFICATION
INPUT
The_Ring: Ring
OUTPUT
The_Ring: Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
   OPERATOR Rotate_To_Mark
SPECIFICATION
INPUT
      The_Ring : Ring
OUTPUT
```

```
The_Ring : Ring
EXCEPTIONS
          Overflow, Underflow, Rotate_Error
   OPERATOR Is_Equal
SPECIFICATION
INPUT
Left : Ring,
Right : Ring
      OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   OPERATOR Extent_Of SPECIFICATION
      INPUT
The_Ring : Ring
OUTFUT
Result : Natural
      EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR Is_Empty
SPECIFICATION
      INPUT
The_Ring : Ring
OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR TOP_Of
SPECIFICATION
INPUT
      The_Ring : Ring
OUTPUT
Result : Item
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR At_Mark
SPECIFICATION
      INPUT
The_Ring : Ring
OUTPUT
         Result : Boolean
      EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR Iterate SPECIFICATION
      GENERIC
Process: PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
INPUT
Over_The_Ring : Ring
FYCEDURANC
      EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   EMD
IMPLEMENTATION ADA
Ring_Sequential_Bounded_Managed_Iterator
```

RING SEQUENTIAL BOUNDED MANAGED NONITERATOR

generic type Item is private; package Ring_Sequential_Bounded_Managed_Noniterator is			
<pre>type Ring(The_Size : Positive) is limited private;</pre>			
type Direction is (Forward, Backward);			
procedure Copy	(From_The_Ring	: in	
Ring;	To_The_Ring	: in out	
Ring); procedure Clear	(The_Ring	: in out	
Ring); procedure Insert	(The_Item	: in	
Item;	In_The_Ring	: in out	
Ring); procedure Pop	(The_Ring	: in out	
Ring);	(The_Ring	: in out	
procedure Rotate Ring;			
Direction);	In_The_Direction		
procedure Mark Ring):	(The_Ring	: in out	
procedure Rotate_To_Mark Ring);	(The_Ring	: in out	

```
function Is_Equal (Left : in Ring; Right : in Ring) return

Boolean;
function Extent_Of (The_Ring : in Ring) return

Ratural;
function Is_Empty (The_Ring : in Ring) return

Boolean;
function Top_Of (The_Ring : in Ring) return

Item;
function At_Mark (The_Ring : in Ring) return

Boolean;
Overflow : exception;
Underflow : exception;
Rotate_Error : exception;

Private
type Items is array(Positive range <>) of Item;
type Ring(The_Size : Positive) is
record
The_Top : Natural := 0;
The_Back : Natural := 0;
The_Hark : Natural := 0;
The_Items : Items(1 . . The_Size);
end record;
end Ring_Sequential_Bounded_Managed_Noniterator;
```

RING SEQUENTIAL BOUNDED MANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
-- Serial Number of the American Rights Legend -- Use, duplication, or disclosure is subject to restrictions as set forth in subdivision (b) (3)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

package body
Ring_Sequential_Bounded_Managed_Noniterator is
        begin
   if From_The_Ring.The_Back >
To_The_Ring.The_Size then
        raise Overflow;
   elsif From_The_Ring.The_Back = 0 then
        To_The_Ring.The_Top := 0;
        To_The_Ring.The_Back := 0;
        To_The_Ring.The_Back := 0;
        to_The_Ring.The_Mark := 0;
   else
                 else
To_The_Ring.The_Items(1 ...
From_The_Ring.The_Back) :=
From_The_Ring.The_Items(1 ...
From_The_Ring.The_Local
From_The_Ring.The_Back);
   To_The_Ring.The_Top :=
From_The_Ring.The_Top;
   To_The_Ring.The_Back :=
From_The_Ring.The_Back: =
From_The_Ring.The_Back: =
To_The_Ring.The_Mark: =
From_The_Ring.The_Mark;
end if;
end Copy;
        procedure Clear (The_Ring : in out Ring) is
begin
   The_Ring.The_Top := 0;
   The_Ring.The_Back := 0;
   The_Ring.The_Mark := 0;
and Clear;
       if In_The_Ring.The_Back = In_The_Ring.The_Size
then
                raise Overflow;
elsif In_The_Ring.The_Back = 0 then
In_The_Ring.The_Top := 1;
In_The_Ring.The_Back := 1;
In_The_Ring.The_Mark := 1;
In_The_Ring.The_Items(1) := The_Item;
else
                        In_The_Ring.The_Items
((In_The_Ring.The_Top + 1) ...
(In_The_Ring.The_Back + 1)) :=
                            In_The_Ring.The_Items(In_The_Ring.The_Top
:= The_Item;
In_The_Ring.The_Back :=
In_The_Ring.The_Back + 1;
    if In_The_Ring.The_Mark >=
In_The_Ring.The_Top then
    In_The_Ring.The_Mark :=
In_The_Ring.The_Mark + 1;
       end if;
end Insert;
        procedure Pop(The_Ring : in out Ring) is
                if The_Ring.The_Back = 0 then
                if The Ring The Back = 0 then
    raise Underflow;
elsif The Ring The Back = 1 then
    The Ring The Top := 0;
    The Ring The Back := 0;
    The Ring The Mark := 0;
                else
The_Ring.The_Items(The_Ring.The_Top ...
The_Back - 1)) :=
The_Ring.The_Leas.\.(The_Ring.The_Back - 1)\) :=
The_Ring.The_Items((The_Ring.The_Top + 1)\). The_Ring.The_Back);
The_Ring.The_Back := The_Ring.The_Back - 1;
if The_Ring.The_Top > The_Ring.The_Back
```

```
if The_Ring.The_Top = The_Ring.The_Mark
 then
                              The_Ring.The_Mark := 1;
                        The_Ring.The_Top := 1;
                  else
                        if The_Ring.The_Mark > The_Ring.The_Top
                              The Ring. The Mark :=
            end if;
end if;
end if;
epp;
The_Ring.The_Mark - 1;
      end Pop;
      procedure Rotate (The_Ring
                                 (The_Ring : in out Ring;
In_The_Direction : in
Direction) is
      begin
           in
  if The_Ring.The_Back = 0 then
    raise Rotate_Error;
elsif In_The_Direction = Forward then
    The_Ring.The_Top := The_Ring.The_Top + 1;
    if The_Ring.The_Top > The_Ring.The_Back
then
                  The_Ring.The_Top := 1;
end if;
            end 1;
else
   The_Ring.The_Top := The_Ring.The_Top - 1;
if The_Ring.The_Top = 0 then
   The_Ring.The_Top := The_Ring.The_Back;
            end if;
      end Rotate:
      procedure Mark (The_Ring : in out Ring) is begin
            The_Ring.The_Mark := The_Ring.The_Top;
      procedure Rotate_To_Mark (The_Ring : in out Ring)
      begin
            The_Ring.The_Top := The_Ring.The_Mark;
      end Rotate_To_Mark;
     modified by Tuan Nguyen
    10 January 1996
adding procedures to replace functions
      procedure Is_Equal (Left
                                     (Left : in Ring;
Right : in Ring;
Result : out Boolean) is
      Result := Is_Equal(Left,Right);
end Is_Equal;
      Result := Extent_Of(The_Ring);
      end Extent_Of;
     procedure Is_Empty (The_Ring : in Ring;
Result : out Boolean) is
     begin
      Result := Is_Empty(The_Ring);
end Is_Empty;
                                    (The_Ring : in Ring;
Result : out Item) is
      procedure Top_Of
     begin
            Result := Top_Of(The_Ring);
      end Top_Of;
     procedure At_Mark (The_Ring : in Ring;
Result : out Boolean) is
      Result := At_Mark(The_Ring); end At_Mark;
    end of modification
      function Is_Equal (Left : in Ring; Right : in Ring) return Boolean
           Left_Index : Natural := Left.The_Top;
Right_Index : Natural := Right.The_Top;
begin
    if Left.The_Back /= Right.The_Back then
        return False;
    elsif Left.The_Items(Left_Index) /=
Right.The_Items(Right_Index) then
        return False;
    elsif (Left.The_Mark = Left_Index) and then
        (Right.The_Mark /= Right_Index) then
        return False;
else
```

```
when Constraint_Error =>
        return (Left.The_Top = Right.The_Top);
end Is_Equal;
function Extent_Of (The_Ring : in Ring) return
Natural is
begin
    return The_Ring.The_Back;
end Extent_Of;
function Is_Empty (The_Ring : in Ring) return
Boolean is
begin
    return (The_Ring.The_Back = 0);
end Is_Empty;
function Top_Of (The_Ring : in Ring) return Item is
begin
    return The_Ring.The_Items(The_Ring.The_Top);
exception
    when Constraint_Error =>
        raise Underflow;
end Top_Of;
function At_Mark (The_Ring : in Ring) return
Boolean is
begin
    return (The_Ring.The_Top = The_Ring.The_Mark);
end Ring_Sequential_Bounded_Managed_Noniterator;
```

RING SEQUENTIAL BOUNDED MANAGED NONITERATOR

```
TYPE Ring_Sequential_Bounded_Managed_Noniterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_Ring: Ring
OUTPUT
To_The_Ring: Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
OPERATOR Clear
SPECIFICATION
INPUT
The_Ring: Ring
OUTPUT
The_Ring: Ring
OUTPUT
The_Ring: Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
OPERATOR Insert
SPECIFICATION
INPUT
The_Item: Item,
In_The_Ring: Ring
OUTPUT
In_The_Ring: Ring
OUTPUT
The_Item: Ring
OUTPUT
In_The_Ring: Ring
OUTPUT
OPERATOR Insert
SPECIFICATION
INPUT
The_Item: Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
OPERATOR Pop
SPECIFICATION
INPUT
The_Ring: Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
OPERATOR Pop
SPECIFICATION
INPUT
The_Ring: Ring
OUTPUT
The_Ring: Ring
OUTPUT
The_Ring: Ring
OUTPUT
The_Ring: Ring
OUTPUT
```

```
The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END

OPERATOR Rotate
SPECIFICATION
INPUT
The_Ring : Ring,
In_The_Direction : Direction
OUTFOT
The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END

OPERATOR Mark
SPECIFICATION
INPUT
The_Ring : Ring
OUTFUT
The_Ring : Ring
OUTFUT
The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END

OPERATOR Rotate_To_Mark
SPECIFICATION
INFUT
The_Ring : Ring
OUTFUT
The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END

END

END

IMPLEMENTATION ADA
Ring_Sequential_Bounded_Managed_Noniterator
END
```

RING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

<pre>generic type Item is private; package Ring_Sequential_Unbounded_Managed_Iterator is</pre>			
type Ring is limited private;			
type Direction is (Forward, Backward);			
procedure Copy	(From_The_Ring : in		
Ring;	To_The_Ring : in out		
Ring); procedure Clear	(The_Ring : in out		
Ring); procedure Insert	(The_Item : in		
Item;	In_The_Ring : in out		
Ring); procedure Pop	(The_Ring : in out		
Ring); procedure Rotate	(The_Ring : in out		
Ring;	In_The_Direction : in		
Direction); procedure Mark	(The_Ring : in out		
Ring); procedure Rotate_To_Mark			
Ring);	\1110_x1119		
modified by Tuan Nguyen 10 January 1996 adding procedures to replace functions			
Rigi	t : in Ring; ht : in Ring; ult : out Boolean);		
procedure Extent_Of (The			
procedure Is_Empty (The			

```
procedure Top_Of (The_Ring : in Ring; Result : out Item);
(The_Ring : in Ring; Result : out Item);
(The_Ring : in Ring; Result : out Boolean);

-- end of modification

function Is_Equal (Left : in Ring; Right : in Ring) return

Boolean;
function Extent_Of (The_Ring : in Ring) return

Natural;
function Is_Empty (The_Ring : in Ring) return

Boolean;
function Top_Of (The_Ring : in Ring) return

Item;
function At_Mark (The_Ring : in Ring) return

Boolean;
generic
with procedure Process (The_Item : in Item;
Continue : out

Boolean);
procedure Iterate (Over_The_Ring : in Ring);

Overflow : exception;
Underflow : exception;
Underflow : exception;
Frotate_Error : exception;
The_Mark : Structure;
The_Mark : Structure;
end record;
end Ring_Sequential_Unbounded_Managed_Iterator;
```

RING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
---
with Storage_Manager_Sequential;
package body Ring_Sequential_Unbounded_Managed_Iterator
is
      type Node is
            record
                   Previous : Structure;
The_Item : Item;
Next : Structure;
            end record:
      procedure Free (The_Node : in out Node) is begin
            The Node Previous := null:
      end Free;
      Structure) is
            The_Node.Next := To_Next;
      end Set Next:
      function Next_Of (The_Node : in Node) return
Structure is
      begin
      return The_Node.Next;
end Next_Of;
      package Node_Manager is new
Storage_Manager_Sequential
                                                      (Item
Node.
                                                       Pointer
Structure.
                               Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
begin
            inf To_The_Ring.The_Top /= null then
   To_The_Ring.The_Top.Previous.Next := null;
   Node_Manager.Free(To_The_Ring.The_Top);
            if From_The_Ring.The_Top = null then
    To_The_Ring.The_Mark := null;
            else
                  To_The_Ring.The_Top :=
Node_Manager.New_Item;
To_The_Ring.The_Top.The_Item :=
From_Index.The_Item;
                  if From The Ring. The Top;
if From The Ring. The Mark = From Inde
To The Ring. The Mark := To Index;
end if;
                                                                   Index then
                  From_Index := From_Index.Next;
while From_Index /= From_The_Ring.The_Top
loop
                        To Index.Next := Node Manager.New Item:
                        To_Index.Next.Previous := To_Index;
To_Index.Next.The_Item :=
From_Index.The_Item;
                        To Index := To Index.Next:
                        if From_The_Ring.The_Mark = From_Index
then
                              To_The_Ring.The_Mark := To_Index;
           end if;

From_Index := From_Index.Next;

end loop;

To_The_Ring.The_Top.Previous := To_Index;

To_Index.Next := To_The_Ring.The_Top;

end if;
      exception
           when Storage_Error => raise Overflow;
      procedure Clear (The Ring : in out Ring) is
```

```
if The Ring.The Top /= null then
                  The Ring. The Top. Previous. Next := null;
Node Manager. Free(The Ring. The Top);
The Ring. The Mark := null;
             end if
       end Clear;
      begin
Node_manager.New_item;
In_The_King.The_Top.Previous :=
In_The_Ring.The_Top.The_Item := The_Item;
In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top;
In_The_Ring.The_Mark :=
In_The_Ring.The_Top;
else
Temporary_Node := Node_Manager.New_Item;
Temporary_Node.Previous :=
In_The_Ring.The_Top.Previous;
Temporary_Node.The_Item := The_Item;
Temporary_Node.Next := In_The_Ring.The_Top;
In_The_Ring.The_Top := Temporary_Node;
In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top:Next:Frevious :=
In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The_Top;
            end if:
       exception
when Storage_Error =>
raise Overflow;
      end Insert:
      procedure Pop(The_Ring : in out Ring) is
   Temporary_Node : Structure;
            Temporary_Node := The_Ring.The_Top;
if The_Ring.The_Top = The_Ring.The_Top.Next
then
                  The_Ring.The_Top := null;
The_Ring.The_Mark := null;
            else
                  The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Previous;
if The_Ring.The_Mark = The_Ring.The_Top
                        The_Ring.The_Mark :=
The_Ring.The_Top.Next;
end if;
The_Ring.The_Top := The_Ring.The_Top.Next;
end if;
            Temporary_Node.Next := null;
            Node_Manager.Free(Temporary_Node);
      exception
when Constraint_Error =>
raise Underflow;
      end Pon.
      procedure Rotate (The_Ring : in out Ring;
In_The_Direction : in
Direction) is
      begin

if In_The_Direction = Forward then

The_Ring.The_Top := The_Ring.The_Top.Next;
The_Ring.The_Top :=
The_Ring.The_Top.Previous;
end if;
      exception
when Constraint_Error =>
                  raise Rotate Error;
      end Rotate:
      procedure Mark (The_Ring : in out Ring) is begin
            The_Ring.The_Mark := The_Ring.The_Top;
      end Mark
      procedure Rotate_To_Mark (The_Ring : in out Ring)
      begin
      The_Ring.The_Top := The_Ring.The_Mark;
end Rotate_To_Mark;
     modified by Tuan Nguyen
10 January 1996
adding procedures to replace functions
      procedure Is_Equal (Left
                                      Right
```

```
when Constraint_Error =>
    return (Left.The_Top = Right.The_Top);
end Is_Equal;
                                  Result : out Boolean) is
      begin
      Result := Is_Equal(Left,Right);
end Is_Equal;
                                                                                              function Extent_Of (The_Ring : in Ring) return
                                                                                        Natural is
Count : Natural := 0;
Index : Structure := The_Ring.The_Top;
      Result := Extent_Of(The_Ring);
                                                                                                   Index := Index.Next;
      end Extent_Of;
                                                                                                   Count := Count + 1;
while Index /= The_Ring.The_Top loop
Count := Count + 1;
Index := Index.Next;
      procedure Is_Empty (The_Ring : in Ring;
Result : out Boolean) is
                                                                                                   end loop;
return Count;
           Result := Is_Empty(The_Ring);
      end Is_Empty;
                                                                                              exception
when Constraint_Error =>
return 0;
end Extent_Of;
                                 (The_Ring : in Ring;
Result : out Item) is
      procedure Top_Of
     begin
           Result := Top_Of(The_Ring);
                                                                                              function Is_Empty (The_Ring : in Ring) return
      end Top_Of;
                                                                                        Boolean is
     procedure At_Mark (The_Ring : in Ring;
Result : out Boolean) is
                                                                                              begin
                                                                                              return (The_Ring.The_Top = null);
end Is_Empty;
           Result := At_Mark(The_Ring);
                                                                                              function Top_Of (The_Ring : in Ring) return Item is
      end At_Mark;
                                                                                             begin
                                                                                                   return The_Ring.The_Top.The_Item;
-- end of modification
                                                                                             return The_name == 
exception when Constraint_Error => 
raise Underflow; 
end Top_Of;
      function Is_Equal (Left : in Ring;
Right : in Ring) return Boolean
           Left_Index : Structure := Left.The_Top;
Right_Index : Structure := Right.The_Top;
                                                                                              function At_Mark (The_Ring : in Ring) return
           if Left_Index.The_Item /= Right_Index.The_Item
                                                                                             begin
                                                                                              return (The_Ring.The_Top = The_Ring.The_Mark);
end At_Mark;
then
           return False;
elsif (Left.The_Mark = Left_Index) and then
  (Right.The_Mark /= Right_Index) then
                                                                                        return False;
Continue
begin

if The_Iterator /= null then
Process(The_Iterator.The_Item, Continue);
if Continue then
The_Iterator := The_Iterator.Next;
while not (The_Iterator =

Over_The_Ring.The_Top) loop
Process(The_Iterator.The_Item,
then
                          (Right.The_Mark /= Right_Index) then return False;
                                                                                             exit
The_I\
end loop;
end if;
end if;
end Iterate;

Ring_Ser
                                                                                                                   exit when not Continue;
The_Iterator := The_Iterator.Next;
                      else
   Left_Index := Left_Index.Next;
   Right_Index := Right_Index.Next;
end if;
           end loop;
return (Right_Index = Right.The_Top);
end if;
                                                                                        end Ring_Sequential_Unbounded_Managed_Iterator;
      exception
```

RING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
TYPE Ring_Sequential_Unbounded_Managed_Iterator SPECIFICATION
   GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
         NPUT
From_The_Ring : Ring,
To_The_Ring : Ring
      OUTPUT
      To_The_Ring : Ring EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR Clear
SPECIFICATION
      INPUT
The_Ring : Ring
OUTPUT
      The_Ring : Ring
EXCEPTIONS
        Overflow, Underflow, Rotate_Error
   EMD
  OPERATOR Insert
SPECIFICATION
     INPUT
The_Item : Item,
In_The_Ring : Ring
      In_The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   END
  OPERATOR Pop
SPECIFICATION
     INPUT
The_Ring : Ring
OUTPUT
The_Ring : Ring
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
  OPERATOR Rotate
SPECIFICATION
     The_Ring: Ring,
In_The_Direction: Direction
OUTPUT
The_Ring: Ring
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
  END
  OPERATOR Mark
   SPECIFICATION
INPUT
The_Ring : Ring
  OUTPUT
The_Ring: Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
  OPERATOR Rotate_To_Mark SPECIFICATION
     INPUT
The_Ring : Ring
OUTPUT
```

```
The_Ring : Ring
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   EMD
   OPERATOR IS_Equal
SPECIFICATION
INPUT
Left: Ring,
Right: Ring
      OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR Extent_Of
SPECIFICATION
INPUT
The_Ring : Ring
      OUTPUT
Result : Natural
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR IS_Empty
SPECIFICATION
INPUT
The_Ring : Ring
OUTPUT
         Result : Boolean
      EXCEPTIONS
Overflow, Underflow, Rotate_Error
   END
   OPERATOR Top_Of
SPECIFICATION
     INPUT
The_Ring : Ring
OUTPUT
Result : Item
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR At_Mark
SPECIFICATION
INPUT
The_Ring : Ring
OUTPUT
      Result : Boolean EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR Iterate
   SPECIFICATION
      GENERIC
Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
INPUT
  INPUT
Over_The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
EMPL
IMPLEMENTATION ADA
Ring_Sequential_Unbounded_Managed_Iterator
```

RING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
generic
   type Item is private;
package Ring_Sequential_Unbounded_Managed_Noniterator
is
     type Ring is limited private;
     type Direction is (Forward, Backward);
procedure Copy
Ring;
                                    (From_The_Ring
                                                          : in
                                                          : in out
                                     To_The_Ring
Ring);
procedure Clear
Ring);
                                                          : in out
                                    (The_Ring
procedure Insert
Item;
                                                          : in
                                     (The_Item
                                     In_The_Ring
                                                          · in out
Ring);
procedure Pop
Ring);
                                                          : in out
                                    (The_Ring
procedure Rotate
Ring;
                                    (The_Ring
                                                          : in out
                                     In_The_Direction : in
Direction);
procedure Mark
Ring);
                                                          : in out
                                    (The_Ring
procedure Rotate_To_Mark (The_Ring
Ring);
                                                          : in out
-- modified by Tuan Nguyen
-- 10 January 1996
-- adding procedures to replace functions
     procedure Is_Equal (Left
                                          : in Ring;
: in Ring;
: out Boolean);
                               Right
Result
```

```
procedure Extent_Of (The_Ring : in Ring;
       procedure Extent_Of The_Ring : in Ring;
procedure Is_Empty (The_Ring : in Ring;
Result : out Boolean);
      Result : out Boolean);
(The_Ring : in Ring;
Result : out Item);
procedure At_Mark (The_Ring : in Ring;
Result : out Boolean);
 -- end of modification
       function Is_Equal (Left Right
                                                 : in Ring;
: in Ring) return
 Boolean;
       function Extent_Of (The_Ring : in Ring) return
 Natural:
       function Is_Empty (The_Ring : in Ring) return
 Boolean; function Top_Of (The_Ring : in Ring) return
 Item;
function At_Mark (The_Ring : in Ring) return
 Boolean;
       Overflow : exception;
Underflow : exception;
Rotate_Error : exception;
private
type Node;
type Structure is access Node;
type Ring is
record
The_Top : Structure;
The_Mark : Structure;
record;
 end record;
end Ring_Sequential_Unbounded_Managed_Noniterator;
```

RING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
                       "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to

-- restrictions as set forth in subdivision (b) (3)
(ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body
Ring_Sequential_Unbounded_Managed_Noniterator is
       type Node is
              record
                       Previous : Structure;
The_Item : Item;
Next : Structure;
              end record:
       procedure Free (The_Node : in out Node) is begin
              The Node.Previous := null;
       procedure Set_Next (The_Node : in out Node;
To_Next : in Structure) is
       begin
The_Node.Next := To_Next;
end Set_Next;
       function Next_Of (The_Node : in Node) return
Structure is
       begin
              return The_Node.Next;
       end Next_Of;
package Node_Manager is new
Storage_Manager_Sequential
                                                                 (Item
                                                                                       =>
Node.
                                                                  Pointer
Structure,
                                     Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
       From_The_Ring.The_Top;
    To_Index : Structure;
       begin
              in
if To_The_Ring.The_Top /= null then
   To_The_Ring.The_Top.Previous.Next := null;
   Node_Manager.Free(To_The_Ring.The_Top);
end if;
              end if;
if From_The_Ring.The_Top = null then
   To_The_Ring.The_Mark := null;
              else To_The_Ring.The_Top :=
Node_Manager.New_Item;
    To_The_Ring.The_Top.The_Item :=
From_Index.The_Item;
To_Index := To_The_Ring.The_Top;
if From_The_Ring.The_Mark = From_Index then
    To_The_Ring.The_Mark := To_Index;
end if;
From_Index := From_Index;
                     end if;
From_Index := From_Index.Next;
while From_Index /= From_The_Ring.The_Top
loop
                            To_Index.Next := Node_Manager.New_Item;
To_Index.Next.Previous := To_Index;
To_Index.Next.The_Item :=
From_Index.The_Item;
                             To_Index := To_Index.Next;
if From_The_Ring.The_Mark = From_Index
 then
              To_The_Ring.The_Mark := To_Index;
end if;
    From_Index := From_Index.Next;
end loop;
To_The_Ring.The_Top.Previous := To_Index;
To_Index.Next := To_The_Ring.The_Top;
end if;
sption
       exception
when Storage_Error =>
raise Overflow;
        end Copy;
        procedure Clear (The_Ring : in out Ring) is
```

```
in
if The_Ring.The_Top /= null then
The_Ring.The_Top.Frevious.Next := null;
Node_Manager.Free(The_Ring.The_Top);
The_Ring.The_Mark := null;
             end Clear;
            begin
if In_The_Ring.The_Top = null then
In_The_Ring.The_Top :=
Node_Manager.New_Item;
In_The_Ring.The_Top.Previous :=
In_The_king.The_top:
In_The_Ring.The_Top.The_Item := The_Item;
In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top:
In_The_Ring.The_Mark :=
 In_The_Ring.The_Top;
                         Temporary_Node := Node_Manager.New_Item;
Temporary_Node := Node_Manager.New_Item;
Temporary_Node.Previous :=
In_The_Ring.The_Top.Previous;
Temporary_Node.The_Item := The_Item;
Temporary_Node.Next := In_The_Ring.The_Top;
In_The_Ring.The_Top := Temporary_Node;
In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The
 In_The_Ring.The_Top;
end if;
             exception
                         when Storage_Error => raise Overflow;
            procedure Pop(The_Ring : in out Ring) is Temporary_Node : Structure;
                         In
Temporary_Node := The_Ring.The_Top;
if The_Ring.The_Top = The_Ring.The_Top.Next
                                     The_Ring.The_Top := null;
The_Ring.The_Mark := null;
                         else
 The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
The_Ring.The_Top.Next.Previous :=
 The_Ring.The_Top.Frevious;
if The_Ring.The_Mark = The_Ring.The_Top
                                                 The_Ring.The_Mark :=
 The_Ring.The_Top.Next;
end if;
The_Ring.The_Top := The_Ring.The_Top.Next;
end if;
                         Temporary_Node.Next := null;
Node_Manager.Free(Temporary_Node);
              exception
when Constraint_Error =>
                                     raise Underflow;
              end Pop;
             Direction) is
              begin
   if In_The_Direction = Forward then
        The_Ring.The_Top := The_Ring.The_Top.Next;
 else The_Ring.The_Top :=
The_Ring.The_Top.Previous;
end if;
              exception
when Constraint_Error =>
raise Rotate_Error;
              end Rotate:
              procedure Mark (The_Ring : in out Ring) is begin
                           The_Ring.The_Mark := The_Ring.The_Top;
              end Mark:
             procedure Rotate_To_Mark (The_Ring : in out Ring)
              begin
              The_Ring.The_Top := The_Ring.The_Mark;
end Rotate_To_Mark;
            modified by Tuan Nguyen
10 January 1996
adding procedures to replace functions
                                                                                                         : in Ring;
: in Ring;
              procedure Is_Equal (Left
```

Right

```
Result : out Boolean) is
     begin
           Result := Is_Equal(Left,Right);
      end Is_Equal;
     begin
     Result := Extent_Of(The_Ring);
end Extent_Of;
     Result := Is_Empty(The_Ring);
end Is_Empty;
                              (The_Ring : in Ring;
Result : out Item) is
     procedure Top_Of
     begin
     Result := Top_Of(The_Ring);
end Top_Of;
     Result := At_Mark(The_Ring);
      end At_Mark;
-- end of modification
     function Is_Equal (Left : in Ring; Right : in Ring) return Boolean
is
          Left_Index : Structure := Left.The_Top;
Right_Index : Structure := Right.The_Top;
     begin
if Left_Index.The_Item /= Right_Index.The_Item
then
          return False;
elsif (Left.The_Mark = Left_Index) and then
(Right.The_Mark /= Right_Index) then
return False;
else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
    while Left_Index /= Left.The_Top loop
    if Left_Index.The_Item /=
Right_Index.The_Item then
    return False;
    elsif (Left.The_Mark = Left_Index) and
then
```

```
(Right.The_Mark /= Right_Index) then return False;
                       else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
           end if;
end loop;
return (Right_Index = Right.The_Top);
end if;
antic;
     exception
when Constraint_Error =>
return (Left.The_Top = Right.The_Top);
      end Is_Equal;
     function Extent_Of (The_Ring : in Ring) return
Natural is
Count : Natural := 0;
Index : Structure := The_Ring.The_Top;
     begin
           in
Index := Index.Next;
Count := Count + 1;
while Index /= The_Ring.The_Top loop
    Count := Count + 1;
    Index := Index.Next;
end loop;
return Count;
     exception
when Constraint_Error =>
     return 0;
end Extent_Of;
     function Is_Empty (The_Ring : in Ring) return
Boolean is
     begin
     return (The_Ring.The_Top = null);
end Is_Empty;
     function Top_Of (The_Ring : in Ring) return Item is
     begin
return The_Ring.The_Top.The_Item;
     exception
when Constraint_Error =>
raise Underflow;
end Top_Of;
     function At_Mark (The_Ring : in Ring) return
Boolean is
     begin
    return (The_Ring.The_Top = The_Ring.The_Mark);
end At_Mark;
end Ring_Sequential_Unbounded_Managed_Noniterator;
```

RING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
TYPE Ring_Sequential_Unbounded_Managed_Noniterator SPECIFICATION
   PECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
INPUT
From_The_Ring: Ring,
To_The_Ring: Ring
OUTPUT
       OUTPUT
To_The_Ring : Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
   OPERATOR Clear
SPECIFICATION
INPUT
The_Ring: Ring
OUTPUT
The_Ring: Ring
       EXCEPTIONS
           Overflow, Underflow, Rotate_Error
   OPERATOR Insert
SPECIFICATION
INPUT
The Item: Item,
In_The_Ring: Ring
OUTPUT
In_The_Ring: Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
   OPERATOR Pop
SPECIFICATION
      PECIFICATION
INPUT
The_Ring : Ring
OUTPUT
The_Ring : Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
    OPERATOR Rotate
SPECIFICATION
      The_Ring : Ring,
In_The_Direction : Direction
       OUTPUT
       The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
    OPERATOR Mark
SPECIFICATION
      INPUT
The_Ring : Ring
       OUTPUT
The_Ring : Ring
EXCEPTIONS
          Overflow, Underflow, Rotate_Error
   END
```

```
OPERATOR Rotate_To_Mark SPECIFICATION
      TRECTION
TINPUT
The_Ring : Ring
OUTPUT
The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   OPERATOR Is_Equal SPECIFICATION
      INPUT
         NPUT
Left : Ring,
Right : Ring
      OUTTRITT
      Result : Boolean
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   END
   OPERATOR Extent_Of SPECIFICATION
      INPUT
The_Ring : Ring
OUTPUT
      Result : Natural
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   OPERATOR IS_Empty
SPECIFICATION
INPUT
      INPUT
The_Ring : Ring
OUTPUT
Result : Boolean
EXCEPTIONS
   Overflow, Underflow, Rotate_Error
   OPERATOR Top_Of
SPECIFICATION
INPUT
The_Ring : Ring
      OUTPUT
      Result : Item
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   END
   OPERATOR At_Mark
SPECIFICATION
      PECIFICATION
INPUT
The_Ring : Ring
OUTPUT
Result : Boolean
EXCEPTIONS
          Overflow, Underflow, Rotate_Error
END
IMPLEMENTATION ADA
Ring_Sequential_Unbounded_Managed_Noniterator
END
```

RING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
generic
 generic
type Item is private;
package Ring_Sequential_Unbounded_Unmanaged_Iterator is
         type Ring is limited private;
         type Direction is (Forward, Backward);
                                                      (From_The_Ring
                                                                                       : in
        procedure Copy
 Ring:
                                                       To_The_Ring
                                                                                       : in out
procedure Clear
Ring);
procedure Insert
Item;
                                                      (The_Ring
                                                                                       : in out
                                                      (The_Item
                                                                                       : in
                                                       In_The_Ring
 Ring);
procedure Pop
Ring);
                                                                                       : in out
                                                       (The_Ring
procedure Rotate
                                                       (The_Ring
                                                                                       : in out
                                                       In_The_Direction : in
 Direction);
procedure Mark (The_Ring
Ring);
procedure Rotate_To_Mark (The_Ring
Ring);
                                                                                       : in out
                                                                                       · in out
 -- modified by Tuan Nguyen
-- 10 January 1996
-- adding procedures to replace functions

        procedure Is_Equal
        (Left ring)
        : in Ring;

        Right result
        : out Boolean);

        procedure Extent_Of (The_Ring result
        : in Ring;

        Result
        : out Natural);

        procedure Is_Empty
        (The_Ring ring);
```

```
Result : out Boolean);
(The_Ring : in Ring;
Result : out Item);
(The_Ring : in Ring;
Result : out Boolean);
      procedure Top_Of
     procedure At_Mark
-- end of modification
                                                : in Ring;
: in Ring) return
      function Is_Equal (Left
                                   Right
Boolean;
function Extent_Of (The_Ring : in Ring) return
Natural:
Natural;
function Is_Empty (The_Ring : in Ring) return
Boolean;
function Top_Of (The_Ring : in Ring) return
Item; function At_Mark (The_Ring : in Ring) return
Boolean;
           with procedure Process (The_Item : in Item;
Continue : out
     procedure Iterate (Over_The_Ring : in Ring);
                        : exception;
      Underflow
                         : exception;
      Rotate_Error : exception;
private
      rate
type Node;
type Structure is access Node;
type Ring is
record
The_Top : Structure;
The_Mark : Structure;
end record;
end Ring_Sequential_Unbounded_Unmanaged_Iterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

 --
package body
Ring_Sequential_Unbounded_Unmanaged_Iterator is
      type Node is
                     Previous : Structure;
The_Item : Item;
Next : Structure;
             end record;
procedure Copy (From_The_Ring : in Ring;
To_The_Ring : in out Ring) is
From_Index : Structure :=
From_The_Ring.The_Top;
To_Index : Structure;
      begin

if From_The_Ring.The_Top = null then

if From_The_Ring.The_Top := null;
                    To_The_Ring.The_Top := null;
To_The_Ring.The_Mark := null;
             else
To_The_Ring.The_Top := new Node'(Previous
=> null.
                                                                          The_Item
 => From_Index.The_Item,
                                                                          Next
 => null);
                    To_Index := To_The_Ring.The_Top;
if From_The_Ring.The_Mark = From_Index then
    To_The_Ring.The_Mark := To_Index;
                    end if;
From_Index := From_Index.Next;
while From_Index /= From_The_Ring.The_Top
1000
                          To Index.Next := new Node'(Previous =>
                                                                       The Item =>
From_Index.The_Item,
                                                                       Next
null):
                           To_Index := To_Index.Next;
if From_The_Ring.The_Mark = From_Index
                           To_The_Ring.The_Mark := To_Index; end if;
             end if;
From_Index := From_Index.Next;
end loop;
To_The_Ring.The_Top.Previous := To_Index;
To_Index.Next := To_The_Ring.The_Top;
end if;
       exception when Storage_Error =>
                  raise Overflow;
       end Copy;
      procedure Clear (The_Ring : in out Ring) is begin
             The_Ring := Ring'(The_Top => null,
The_Mark => null);
       end Clear:
      if In_The_Ring.The_Top = null then
In_The_Ring.The_Top := new Node'(Previous
=> The Item.
                                                                          Next
                    In_The_Ring.The_Top.Previous :=
In_The_Ring.The_Top;
In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top;
In_The_Ring.The_Mark :=
 In_The_Ring.The_Top;
             else
In_The_Ring.The_Top :=
new Node' (Previous =>
In_The_Ring.The_Top.Previous,
The_Item => The_Item,
```

```
In_The_Ring.The_Top);
               In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top:Next:Flevious :=
In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The_Top;
          end if:
     exception
when Storage_Error =>
               raise Overflow;
     end Insert;
     procedure Pop(The_Ring : in out Ring) is
          if The_Ring.The_Top = The_Ring.The_Top.Next
               The_Ring.The_Top := null;
The_Ring.The_Mark := null;
          else
The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Previous;
if The_Ring.The_Mark = The_Ring.The_Top
                   The_Ring.The_Mark :=
The_Ring.The_Top.Next;
end if;
The_Ring.The_Top := The_Ring.The_Top.Next;
end if;
     exception
     raise Underflow;
end Pop;
     : in out Ring:
Direction) is
     begin
          if In_The_Direction = Forward then
    The_Ring.The_Top := The_Ring.The_Top.Next;
The_Ring.The_Top :=

The_Ring.The_Top :=

The_Ring.The_Top.Previous;

end if;
     exception
   when Constraint_Error =>
      raise Rotate_Error;
end Rotate;
     procedure Mark (The_Ring : in out Ring) is begin
          The_Ring.The_Mark := The_Ring.The_Top;
     end Mark:
    procedure Rotate_To_Mark (The_Ring : in out Ring)
     The_Ring.The_Top := The_Ring.The_Mark;
end Rotate_To_Mark;
     modified by Tuan Nguyen
10 January 1996
adding procedures to replace functions
     procedure Is_Equal (Left
                                          · in Rings
                              Right
Result
                                         : in Ring;
: out Boolean) is
     begin
          Result := Is_Equal(Left,Right);
     end Is_Equal;
     Result := Extent_Of(The_Ring);
end Extent_Of;
     Result := Is_Empty(The_Ring);
     end Is_Empty;
     procedure Top Of
                              (The_Ring : in Ring;
Result : out Item) is
     begin
     Result := Top_Of(The_Ring);
end Top_Of;
     procedure At_Mark (The_Ring : in Ring;
Result : out Boolean) is
          Result := At_Mark(The_Ring);
     end At_Mark;
     end of modification
```

```
function Is_Equal (Left : in Ring;
Right : in Ring) return Boolean
is
               Left_Index : Structure := Left.The_Top;
Right_Index : Structure := Right.The_Top;
       begin
               if Left_Index.The_Item /= Right_Index.The_Item
then
               return False;
elsif (Left.The_Mark = Left_Index) and then
(Right.The_Mark /= Right_Index) then
return False;
               else
                      e

Left_Index := Left_Index.Next;

Right_Index := Right_Index.Next;

while Left_Index /= Left.The_Top loop

if Left_Index.The_Item /=
Right_Index.The_Item then
return False;
elsif (Left.The_Mark = Left_Index) and
then
                                   (Right.The_Mark /= Right_Index) then return False;
                             return rass,
else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
end if;
                      end loop;
return (Right_Index = Right.The_Top);
               end if;
       exception
  when Constraint_Error =>
      return (Left.The_Top = Right.The_Top);
        function Extent_Of (The_Ring : in Ring) return
Natural is
Count: Natural := 0;
Index: Structure := The_Ring.The_Top;
       begin
              in
Index := Index.Next;
Count := Count + 1;
while Index /= The_Ring.The_Top loop
    Count := Count + 1;
Index := Index.Next;
end loop;
return Count;
```

```
exception
              when Constraint_Error =>
       return 0;
end Extent_Of;
       function Is_Empty (The_Ring : in Ring) return
 Boolean is
       begin
       return (The_Ring.The_Top = null);
end Is_Empty;
       function Top_Of (The_Ring : in Ring) return Item is
       begin
return The_Ring.The_Top.The_Item;
       exception
when Constraint_Error =>
raise Underflow;
end Top_Of;
       function At_Mark (The_Ring : in Ring) return
 Boolean is
       begin
    return (The_Ring.The_Top = The_Ring.The_Mark);
end At_Mark;
procedure Iterate (Over_The_Ring : in Ring) is
    The_Iterator : Structure :=
Over_The_Ring.The_Top;
    Continue : Boolean;
Continue : Boolean;
begin

if The_Iterator /= null then
Process(The_Iterator.The_Item, Continue);
if Continue then
The_Iterator := The_Iterator.Next;
while not (The_Iterator =

Over_The_Ring.The_Top) loop
Process(The_Iterator.The_Item,
            exit when not Continue;
The_Iterator := The_Iterator.Next;
end loop;
end if;
end if;
Iterator
Continue);
       end Iterate;
end Ring_Sequential_Unbounded_Unmanaged_Iterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
TYPE Ring_Sequential_Unbounded_Unmanaged_Iterator SPECIFICATION
   GENERIC
   GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
From_The_Ring : Ring,
To_The_Ring : Ring
      OUTPUT
To_The_Ring : Ring
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR Clear
   SPECIFICATION
INPUT
The_Ring : Ring
      OUTPUT
The Ring : Ring
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR Insert
SPECIFICATION
      INPUT
The_Item : Item,
In_The_Ring : Ring
      OUTPUT
In The Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   END
   OPERATOR Pop
SPECIFICATION
      INPUT
      The_Ring : Ring
OUTPUT
The_Ring : Ring
  Overflow, Underflow, Rotate_Error
   OPERATOR Rotate
SPECIFICATION
     The_Ring : Ring,
In_The_Direction : Direction
OUTPUT
The_Ring : Ring
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR Mark
   OPERATOR Mark
SPECIFICATION
INPUT
The_Ring: Ring
OUTFUT
The_Ring: Ring
EKCEPTIONS
OUTFUT
LINEARING: LINEARING
         Overflow, Underflow, Rotate_Error
   OPERATOR Rotate_To_Mark
   SPECIFICATION
     INPUT
The_Ring : Ring
OUTPUT
```

```
The_Ring : Ring EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   OPERATOR Is_Equal SPECIFICATION
      INPUT
Left : Ring,
Right : Ring
      OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR Extent_Of
   SPECIFICATION
INPUT
         The_Ring : Ring
      OUTPUT
      Result : Natural
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR IS_Empty SPECIFICATION
      INPUT
The_Ring : Ring
OUTPUT
Result : Boolean
      EXCEPTIONS
Overflow, Underflow, Rotate_Error
  OPERATOR Top_Of
SPECIFICATION
INPUT
The_Ring : Ring
      Result : Item
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
  OPERATOR At_Mark
SPECIFICATION
INPUT
The_Ring : Ring
OUTPUT
      Result : Boolean
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
   OPERATOR Iterate SPECIFICATION
      GENERIC
Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
INPUT
      INPUT
Over_The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
   END
IMPLEMENTATION ADA
Ring_Sequential_Unbounded_Unmanaged_Iterator
```

RING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
generic
type Item is private;
package Ring_Sequential_Unbounded_Unmanaged_Noniterator
is
     type Ring is limited private;
     type Direction is (Forward, Backward);
     procedure Copy
                                     (From_The_Ring
                                                          : in
Ring;
                                                          : in out
                                     To_The_Ring
Ring);
procedure Clear
Ring);
                                    (The_Ring
                                                          : in out
     procedure Insert
                                     (The_Item
                                                          : in
Item;
                                     In_The_Ring
                                                          : in out
procedure Pop
Ring);
procedure Rotate
Ring;
Ring);
                                                          : in out
                                                          : in out
                                     (The_Ring
                                     In The Direction : in
Direction);
procedure Mark (The_Ring
Ring);
procedure Rotate_To_Mark (The_Ring
Ring);
                                                          : in out
    modified by Tuan Nguyen
10 January 1996
     adding procedures to replace functions
                                          : in Ring;
: in Ring;
: out Boolean);
     procedure Is_Equal
                              (Left
                                Right
                               Result
```

```
procedure Extent_Of (The_Ring : in Ring;
Result : out Natur
procedure Is_Empty (The_Ring : in Ring;
                                        (The_Ring : in Ring;
Result : out Natural);
(The_Ring : in Ring;
Result : out Boolean);
(The_Ring : in Ring;
Result : out Item);
(The_Ring : in Ring;
      procedure Top_Of
      procedure At_Mark
                                                        : out Boolean);
                                         Result
-- end of modification
      function Is_Equal (Left Right
                                                   : in Ring;
: in Ring) return
      function Extent_Of (The_Ring : in Ring) return
Natural:
      function Is_Empty (The_Ring : in Ring) return
Boolean;
function Top_Of
                                     (The_Ring : in Ring) return
Item;
function At_Mark (The_Ring : in Ring) return
      Overflow : exception;
Underflow : exception;
Rotate_Error : exception;
private
      type Node;
type Structure is access Node;
type Ring is
record
record
The_Top : Structure;
The_Mark : Structure;
end record;
end Ring_Sequential_Unbounded_Unmanaged_Noniterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
                   "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)

    (11)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

package body
Ring_Sequential_Unbounded_Unmanaged_Noniterator is
      type Node is
            record
                   Previous : Structure;
The_Item : Item;
Next : Structure;
            end record;
     From_The_Ring.The_Top;
    To_Index : Structure;
     To_index

begin

if From_The_Ring.The_Top = null then

To_The_Ring.The_Top := null;

To_The_Ring.The_Mark := null;
            else
To_The_Ring.The_Top := new Node'(Previous
=> null.
                                                                  The_Item
=> From Index.The_Item,
=> null);
                  To_Index := To_The_Ring.The_Top;
if From_The_Ring.The_Mark = From_Index;
To_The_Ring.The_Mark := To_Index;
                                                                   Index then
                  end if:
                  end if;
From_Index := From_Index.Next;
while From_Index /= From_The_Ring.The_Top
1000
                        To Index Next := new Node'(Previous =>
To_Index,
                                                                The_Item =>
From Index.The Item,
                                                                Next
null);
                        To_Index := To_Index.Next;
if From_The_Ring.The_Mark = From_Index
then
                              To_The_Ring.The_Mark := To_Index;
                        end if;
From_Index := From_Index.Next;
            end loop;

end loop;

To_The_Ring.The_Top.Previous := To_Index;

To_Index.Next := To_The_Ring.The_Top;

end if;
      exception
when Storage_Error =>
raise Overflow;
      end Copy:
      procedure Clear (The_Ring : in out Ring) is begin
            The_Ring := Ring'(The_Top => null,
The Mark => null);
      end Clear;
      begin
           if In_The_Ring.The_Top = null then
    In_The_Ring.The_Top := new Node'(Previous)
=> null,
                                                                   The Item
=> The Item.
                  In_The_Ring.The_Top.Previous :=
In_The_Ring.The_Top;
    In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top:
In_The_Ring.The_Mark :=
In_The_Ring.The_Top;
            else
In_The_Ring.The_Top :=
new Node'(Previous =>
Top Previous,
In_The_Ring.The_Top.Previous,
The_Item => The_Item,
```

```
In_The_Ring.The_Top);
In_The_Ring.The_Top.Next.Previous := In_The_Ring.The_Top.Previous.Next := In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The_Top;
           end if;
     exception
when Storage_Error =>
               raise Overflow;
     end Insert;
     procedure Pop(The_Ring : in out Ring) is
     begin
if The_Ring.The_Top = The_Ring.The_Top.Next
then
               The_Ring.The_Top := null;
The_Ring.The_Mark := null;
          else
The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Previous;
if The_Ring.The_Mark = The_Ring.The_Top
                     The_Ring.The_Mark :=
The_Ring.The_Top.Next;
end if;
The_Ring.The_Top := The_Ring.The_Top.Next;
end if;
     exception
     raise Underflow;
end Pop;
     Direction) is
     begin
          in
  if In_The_Direction = Forward then
     The_Ring.The_Top := The_Ring.The_Top.Next;
          else
The_Ring.The_Top :=
The Ring.The_Top :=
The_Ring.The_Top.Previous;
end if;
exception
when Constraint_Error =>
     raise Rotate_Error;
end Rotate;
     procedure Mark (The_Ring : in out Ring) is
     The_Ring.The_Mark := The_Ring.The_Top; end Mark;
     procedure Rotate_To_Mark (The_Ring : in out Ring)
           The Ring. The Top := The Ring. The Mark;
     end Rotate_To_Mark;
-- modified by Tuan Nguyen
-- 10 January 1996
-- adding procedures to replace functions
                                (Left : in Ring;
Right : in Ring;
Result : out Boolean) is
     procedure Is_Equal (Left Right
     Result := Is_Equal(Left,Right);
end Is_Equal;
     Result := Extent_Of(The_Ring);
      end Extent_Of;
     procedure Is_Empty (The_Ring : in Ring;
Result : out Boolean) is
     Result := Is_Empty(The_Ring);
end Is_Empty;
                                (The_Ring : in Ring;
Result : out Item) is
      procedure Top_Of
      begin
           Result := Top_Of(The_Ring);
      end Top_Of;
     procedure At_Mark (The_Ring : in Ring;
Result : out Boolean) is
      Result := At_Mark(The_Ring);
end At_Mark;
      end of modification
```

```
Count: Natural := 0;
    Index: Structure := The_Ring.The_Top;

begin
    Index := Index.Next;
    Count := Count + 1;
    while Index /= The_Ring.The_Top loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
    exception
    when Constraint_Error =>
        return 0;
    end Extent_Of;
    function Is_Empty (The_Ring : in Ring) return

Boolean is
    begin
        return (The_Ring.The_Top = null);
    end Is_Empty;
    function Top_Of (The_Ring : in Ring) return Item is begin
        return The_Ring.The_Top.The_Item;
    exception
    when Constraint_Error =>
        raise Underflow;
    end Top_Of;
    function At_Mark (The_Ring : in Ring) return

Boolean is
    begin
    return (The_Ring.The_Top = The_Ring.The_Mark);
    end At_Mark;
end Ring_Sequential_Unbounded_Unmanaged_Noniterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
{\tt TYPE\ Ring\_Sequential\_Unbounded\_Unmanaged\_Noniterator} \\ {\tt SPECIFICATION}
   GENERIC
    GENERIC
Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
   SPECIFICATION
INPUT
From_The_Ring : Ring,
To_The_Ring : Ring
OUTPUT
TO_The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
   OPERATOR Clear
SPECIFICATION
       INPUT
       The_Ring : Ring
OUTPUT
The_Ring : Ring
       EXCEPTIONS
           Overflow, Underflow, Rotate_Error
    OPERATOR Insert
SPECIFICATION
       INPUT
       INPUT
The_Item : Item,
In_The_Ring : Ring
OUTPUT
In_The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
    END
    OPERATOR Pop
SPECIFICATION
INPUT
           The_Ring : Ring
       OUTPUT
The_Ring : Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
   OPERATOR Rotate
SPECIFICATION
       INPUT
The_Ring : Ring,
In_The_Direction : Direction
       The_Ring : Ring
EXCEPTIONS
Overflow, Underflow, Rotate_Error
    END
    OPERATOR Mark
SPECIFICATION
       INPUT
The_Ring : Ring
OUTPUT
The_Ring : Ring
EXCEPTIONS
           Overflow, Underflow, Rotate_Error
```

```
OPERATOR Rotate_To_Mark
   SPECIFICATION
INPUT
The_Ring : Ring
OUTPUT
      The Ring : Ring
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   EMD
  OPERATOR Is_Equal SPECIFICATION
     INPUT
Left : Ring,
Right : Ring
      OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
  OPERATOR Extent_Of SPECIFICATION
      INPUT
The_Ring : Ring
OUTPUT
      Result : Natural
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
  OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Ring : Ring
OUTPUT
  Result : Boolean
EXCEPTIONS
Overflow, Underflow, Rotate_Error
END
   OPERATOR Top_Of
SPECIFICATION
INPUT
The_Ring : Ring
      OUTPUT
Result : Item
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
   END
  OPERATOR At_Mark SPECIFICATION
      PECIFICATION
INPUT
The_Ring : Ring
OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Underflow, Rotate_Error
EMPLEMENTATION ADA
Ring_Sequential_Unbounded_Unmanaged_Noniterator
```

SETS OBJ3 SPECIFICATIONS

```
obj SET[X :: TRIV] is sort Set .
protecting NAT .

*** constructors

op create : -> Set .-> Set .
op copy :: Set Set -> Set .
op add :: Elt Set -> Set .
op union :: Set Set Set -> Set .
op union :: Set Set Set -> Set .
op intersection :: Set Set Set -> Set .
op intersection :: Set Set Set -> Set .

*** accessors

op isequal :: Set Set -> Bool .
op extentof :: Set -> Bool .
op isamember :: Elt Set -> Bool .
op isamember :: Elt Set -> Bool .
op isapropersubset :: Set Set -> Bool .
op isapropersubset :: Set Set -> Bool .
op itemisinset :-> Set .

*** exceptions

op overflow :-> Set .
op itemisinset :-> Set .

*** variables declaration

var S Sl S2 :: Set .
var E El :: Elt .

*** axioms
```

eq copy(S,S1) = S.

```
eq clear(S) = create .
eq remove(E, create) = itemisnotinset .
eq remove(E, add(E1,S1)) = if E == E1 then S1 else
add(E1,remove(E,S1)) fi .

eq union(S, create,S1) = S .
eq union(S, add(E1,S1),S2) = if isamember(E1,S) then union(S,S1,S2)
else add(E1,union(S,S1,S2)) fi .

eq intersection(S, create,S1) = create .
eq intersection(S, add(E1,S1),S2) = if isamember(E1,S) then
add(E1,intersection(S,S,S2)) else intersection(S,S1,S2) fi .

eq difference(create,S,S1) = S .
eq difference(S,create,S1) = S .
eq difference(S,add(E1,S1),S2) = if isamember(E1,S) then
difference(remove(E1,S),S1,S2) else add(E1,difference(S,S1,S2)) fi .

eq isequal(S,S1) = S == S1 .

eq extentof(create) = 0 .
eq extentof(add(E,S)) = 1 + extentof(S) .

eq isamember(E, create) = false .
eq isamember(E, add(E1,S1)) = E == E1 or isamember(E,S1) .

eq isasubset(create,S) = true .
eq isasubset(add(E,S),S1) = if isamember(E,S1) then
isasubset(S,remove(E,S1)) else false fi .
eq isapropersubset(S,S1) = isasubset(S,S1) and extentof(S1) >
extentof(S) .
```

SET PROFILE CODES

OPERATORS	SIGNATURES	PROFILE CODES
COPY	A B -> B	3211
CLEAR	A -> A	2201
ADD	A B -> B	3211
REMOVE	A B -> B	3211
UNION	ABC->C	4231
INTERSECTION	ABC->C	4231
DIFFERENCE	A B C -> C	4231
IS_EQUAL	A B -> C	330
EXTENT_OF	A -> B	220
IS_EMPTY	A -> B	220
IS_A_MEMBER	A B -> C	330
IS_A_SUBSET	A B -> C	330
IS_A_PROPER_SUBSET	A B -> C	330

SET OF PROFILE: {4231,3211,2201,330,220}

SET SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
generic
    type Item is private;
package Set_Simple_Sequential_Bounded_Managed_Iterator is
       type Set(The_Size : Positive) is limited private;
                                                  (From_The_Set : in
To_The_Set : in
(The_Set : in
       procedure Copy
                                                                                in out Set);
in out Set);
       procedure Clear
procedure Add
                                                   (The_Item
To_The_Set
(The_Item
                                                                                in
                                                                                             Item:
                                                                                in out Set);
in Item;
                                                                                in Item;
in out Set);
       procedure Remove
                                                    From_The_Set :
      procedure Union (Of_The_Set
And_The_Set
And_The_Set
Procedure Intersection (Of_The_Set
And_The_Set
Of_The_Set
And_The_Set
To_The_Set
(Of_The_Set
And_The_Set
To_The_Set
To_The_Set
To_The_Set
                                                                                in
                                                                                             Set:
                                                                                in Set;
in out Set);
                                                                                in
                                                                                             Set:
                                                                                in Set;
in out Set);
                                                                             : in
                                                                                             Set:
       modified by Tuan Nguyen
       20 Aug 95
replacing functions with procedures
                                                                                    : in Set;
: in Set;
: out Boolean);
       procedure Is_Equal
                                                               Right
                                                              Result
(The_Set
Result
(The_Set
                                                                                    cout Boolean);
in Set;
out Natural);
in Set;
cout Boolean);
in Item;
in Set;
out Boolean);
       procedure Extent_Of
       procedure Is_Empty
                                                              Result :
(The_Item :
Of_The_Set :
Result :
       procedure Is_A_Member
```

```
procedure Is_A_Subset
                                                    (Left
                                                                      : in Set;
                                                                         in Set;
out Boolean);
in Set;
                                                     Right
                                                   Result
(Left
      procedure Is_A_Proper_Subset
                                                     Right
                                                                         in Set
                                                     Result
-- end of modification
                                                                    : in Set;
: in Set)
: in Set)
: in Set)
: in Set)
: in Set;
: in Set;
      function Is Equal
                                                  (Left
                                                                                     return Boolean;
                                                  Right
(The_Set
(The_Set
(The_Item
      function Extent_Of
function Is_Empty
function Is_A_Member
                                                                                     return Natural;
                                                                                      return Boolean;
                                                                                     return Boolean:
                                                  Of_The_Set
(Left
      function Is_A_Subset
                                                   Right
                                                                                     return Boolean;
      function Is_A_Proper_Subset (Left
                                                                       in Set:
                                                                     : in Set) return Boolean;
            with procedure Process (The_Item : in Item;
Continue : out Boolean);
      procedure Iterate (Over_The_Set : in Set);
      Overflow : exception; Item_Is_In_Set : exception; Item_Is_Not_In_Set : exception;
      type Items is array(Positive range <>) of Item;
type Set(The_Size : Positive) is
            record
record
    The_Back : Natural := 0;
    The_Items : Items(1 .. The_Size);
    end record;
end Set_Simple_Sequential_Bounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

```
(C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch All Rights Reserved
    Serial Number 0100219
                     "Restricted Rights Legend"
-- Wes, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body Set_Simple_Sequential_Bounded_Managed_Iterator is
      begin
   if From_The_Set.The_Back > To_The_Set.The_Size then
      raise Overflow;
      else
   To_The_Set.The_Items(1 .. From_The_Set.The_Back) :=
        From_The_Set.The_Items(1 .. From_The_Set.The_Back);
   To_The_Set.The_Back := From_The_Set.The_Back;
end if;
end Copy;
      procedure Clear (The_Set : in out Set) is
begin
    The_Set.The_Back := 0;
end Clear;
      for Index in 1 .. To_The_Set.The_Back loop
if The_Item = To_The_Set.The_Items(Index) then
raise Item_Is_In_Set;
end if;
             end loop;
To_The_Set.The_Items(To_The_Set.The_Back + 1) := The_Item;
To_The_Set.The_Back := To_The_Set.The_Back + 1;
      exception
   when Constraint_Error =>
     raise Overflow;
end Add;
      procedure Remove (The_Item : in Item;
    From_The_Set : in out Set) is
              for Index in 1 .. From_The_Set.The_Back loop
    if The_Item = From_The_Set.The_Items(Index) then
        From_The_Set.The_Items(Index .. (From_The_Set.The_Back))
- 1)) :=
                               From_The_Set.The_Items((Index + 1) ...
From_The_Set.The_Back):
From_The_Set.The_Back := From_The_Set.The_Back - 1;
                     return;
end if;
             end li;
end loop;
raise Item_Is_Not_In_Set;
       end Remove:
      procedure Union (Of_The_Set : in Set;
And_The_Set : in Set;
To_The_Set : in out Set) is
             To_Index : Natural;
To_Back : Natural;
       begin
             exit:
                            else
To_Index := To_Index - 1;
end if;
                    end 11;
end loop;
if To_Index = 0 then
   To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
   And_The_Set.The_Items(And_Index);
   To_The_Set.The_Back := To_The_Set.The_Back + 1;
              end if;
end loop;
        exception
              when Constraint Error =>
       raise Overflow;
end Union;
       procedure Intersection (Of_The_Set : in Set;
And_The_Set : in Set;
To_The_Set : in out Set) is
```

```
And_Index : Natural;
       begin
      ease
And_Index := And_Index - 1;
end if;
end loop;
end loop;
eption
   end loop,
exception
when Constraint_Error =>
raise Overflow;
end Intersection;
   And_Index : Natural;
   begin
       And_The_Set.The_Items(And_Index) then
               exit;
else
                And_Index := And_Index - 1;
end if;
            end loop;
if And_Index = 0 then
               end if:
       end loop;
   end loop;
exception
when Constraint_Error =>
raise Overflow;
end Difference;
-- modified by Tuan Nguyen
  20 Aug 95
replacing functions with procedures
                                              : in Set;
: in Set;
   procedure Is_Equal
                                   Right
Result
                                                out Boolean) is
        Result := Is_Equal(Left, Right);
                                  (The_Set
                                              : in Set;
: out Natural) is
   procedure Extent_Of
                                   Result
   begin
   Result := Extent_Of(The_Set);
end Extent_Of;
    procedure Is Empty
                                  (The Set
                                              : in Set;
                                   Result
                                              : out Boolean) is
        Result := Is_Empty(The_Set);
    end Is_Empty;
                                  (The_Item : in Item;
Of_The_Set : in Set;
Result : out Boolean) is
   procedure Is_A_Member
       Result := Is_A_Member(The_Item,Of_The_Set);
    end Is_A_Member;
                                  (Left
Right
                                              : in Set;
: in Set;
: out Boolean) is
   procedure Is_A_Subset
                                   Result
   Result := Is_A_Subset(Left, Right);
end Is_A_Subset;
    procedure Is_A_Proper_Subset (Left
                                              : in Set; : in Set;
                                   Right
Result
                                              : out Boolean) is
   begin
    Result := Is_A_Proper_Subset(Left, Right);
end Is_A_Proper_Subset;
-- end of modification
```

```
for Left_Index in 1 .. Left.The_Back loop
   Right_Index := Right.The_Back;
   while Right_Index > 0 loop
    if Left.The_Items(Left_Index) =
        Right.The_Items(Right_Index) then
                                       exit:
                                else
                       else
    Right_Index := Right_Index - 1;
end if;
end loop;
if Right_Index = 0 then
                       return False;
end if;
        end loop;
return True;
end Is_A_Subset;
        function Is_A_Proper_Subset (Left : in Set;
Right : in Set) return Boolean is
                Right_Index : Natural;
      Right_index : Annual . Left.The_Back loop
    Right_Index := Right.The_Back;
    while Right_Index > 0 loop
        if Left.The_Items(Left_Index) =
            Right.The_Items(Right_Index) then
        exit;
                                       exit;
                               else
Right_Index := Right_Index - 1;
end if;
                       end loop;
if Right_Index = 0 then
                               return False;
                        end if;
       end loop;
return (Left.The_Back < Right.The_Back);
end Is_A_Proper_Subset;
       procedure Iterate (Over_The_Set : in Set) is
   Continue : Boolean;
       begin
       begin
    for The_Iterator in 1 .. Over_The_Set.The_Back loop
        Process(Over_The_Set.The_Items(The_Iterator), Continue);
        exit when not Continue;
end loop;
end Iterate;
end Set_Simple_Sequential_Bounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

```
TYPE Set_Simple_Sequential_Bounded_Managed_Iterator SPECIFICATION
   GENERIC
  GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_Set: Set,
         To_The_Set : Set
      To_The_Set : Set
OUTPUT
To_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Clear
   SPECIFICATION
INPUT
The_Set : Set
      OUTPUT
     The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   EMD
   OPERATOR Add
   SPECIFICATION
     INPUT
The_Item : Item,
To_The_Set : Set
      OUTPUT
      To_The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Remove
SPECIFICATION
INPUT
The_Item : Item,
From_The_Set : Set
OUTPUT
        From_The_Set : Set
      EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   EMD.
   OPERATOR Union
SPECIFICATION
      INPUT
        Of_The_Set : Set,
And_The_Set : Set,
To_The_Set : Set
      CUTPUT
      OUTFUT
To_The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   END
   OPERATOR Intersection SPECIFICATION
      INPUT
        Of_The_Set : Set,
And_The_Set : Set,
To_The_Set : Set
         To_The_
                   _Set : Set
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Difference SPECIFICATION
         Of_The_Set : Set,
And_The_Set : Set,
To_The_Set : Set
      OUTPUT
         Overflow, Item Is_In_Set, Item_Is_Not_In_Set
```

```
END
  OPERATOR IS_Equal SPECIFICATION
     INPUT
Left : Set,
Right : Set
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  FND
  OPERATOR Extent_Of SPECIFICATION
     INPUT
The_Set : Set
OUTPUT
     OUTPUT
Result : Natural
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_Empty
SPECIFICATION
    INPUT
The_Set : Set
OUTPUT
        Result : Boolean
     EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Member
SPECIFICATION
INPUT
The_Item : Item,
Of_The_Set : Set
     OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 OPERATOR IS_A_Subset
SPECIFICATION
INPUT
Left : Set,
Right : Set
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Proper_Subset
SPECIFICATION
INPUT
Left : Set,
        Right : Set
     OUTPUT
RESult: Boolean
EXCEPTION, Item_Is_In_Set, Item_Is_Not_In_Set
  END
  OPERATOR Iterate SPECIFICATION
     GENERIC
         Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
 Over_The_Set : Set

EXCEPTIONS

Overflow, Item_Is_In_Set, Item_Is_Not_In_Set

END
IMPLEMENTATION ADA Set_Simple_Sequential_Bounded_Managed_Iterator
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
type Item is private;
package Set_Simple_Sequential_Bounded_Managed_Noniterator is
      type Set(The_Size : Positive) is limited private;
                                        (From_The_Set : in Set;
To_The_Set : in out Set);
(The_Set : in out Set);
(The_Item : in Item;
To_The_Set : in out Set);
     procedure Copy
     procedure Clear
                                         (The_Item
From_The_Set
(Of_The_Set
      procedure Remove
                                                                 in
                                                                           Item:
                                                                 in out Set);
     procedure Union
                                          And_The_Set
                                                                 in
                                                                           Set;
     To_The_Set
procedure Intersection (Of_The_Set
And_The_Set
                                                                in out Set);
                                                                           Set;
Set;
                                                              : in Set;
: in out Set);
                                        To_The_Set
(Of_The_Set
And_The_Set
                                                                           Set;
Set;
      procedure Difference
                                                              : in Set;
: in out Set);
                                          To_The_Set
     modified by Tuan Nguyen
     20 Aug 95
replacing functions with procedures
                                                                   : in Set;
: in Set;
: out Boolean);
: in Set;
: out Natural);
: in Set;
                                                  (Left
      procedure Is_Equal
                                                   Right
                                                  Result
(The_Set
      procedure Extent_Of
                                                   Result
                                                  (The Set
      procedure Is_Empty
                                                                    : out Boolean);
                                                   Result
```

```
procedure Is_A_Member
                                                  (The_Item : in Item
Of_The_Set : in Set;
Result : out Boo
                                                                        out Boolean);
in Set;
      procedure Is_A_Subset
                                                   (Left
                                                   Right
Result
                                                                       in Set:
                                                                       out Boolean);
in Set;
      procedure Is_A_Proper_Subset
                                                   (Left
                                                    Right
                                                                       in Set:
                                                                     : out Boolean);
-- end of modification
                                                                   : in Set;
: in Set)
: in Set)
: in Set)
: in Item;
      function Is_Equal
                                                 (Left
Right
                                                                                    return Boolean;
                                                 (The_Set
(The_Set
(The_Item
Of_The_Set
      function Extent_Of
function Is_Empty
function Is_A_Member
                                                                                    return Natural:
                                                                                    return Boolean;
                                                                     in Set;
in Set;
in Set)
                                                                                   return Boolean;
     function Is_A_Subset (Left Right function Is_A_Proper_Subset (Left
                                                                                   return Boolean;
                                                                      in Set;
                                                                   : in Set) return Boolean;
                                                  Right
      Overflow
                                  : exception:
     Item_Is_In_Set : exception;
Item_Is_Not_In_Set : exception;
     type Items is array(Positive range <>) of Item;
type Set(The_Size : Positive) is
           record
The_Back : Natural := 0;
The_Items : Items(1 .. The_Size);
end record;
end Set_Simple_Sequential_Bounded_Managed_Noniterator;
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch -- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FRR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body Set_Simple_Sequential_Bounded_Managed_Noniterator is
      begin
             if From_The_Set.The_Back > To_The_Set.The_Size then
                   raise Overflow;
             raise OverLiuw,
else
To_The_Set.The_Items(1 .. From_The_Set.The_Back) :=
From_The_Set.The_Items(1 .. From_The_Set.The_Back);
To_The_Set.The_Back := From_The_Set.The_Back;
      end Copy;
      procedure Clear (The_Set : in out Set) is begin
             The_Set.The_Back := 0;
       end Clear;
      TO_TNe_Set.The_Back := To
exception
   when Constraint_Error =>
        raise Overflow;
end Add;
      procedure Remove (The_Item : in Item;
From_The_Set : in out Set) is
             for Index in 1 .. From_The_Set.The_Back loop
    if The_Item = From_The_Set.The_Items(Index) then
        From_The_Set.The_Items(Index .. (From_The_Set.The_Back))
- 11) :=
                             From_The_Set.The_Items((Index + 1) ...
From_The_Set.The_Back);
From_The_Set.The_Back := From_The_Set.The_Back - 1;
                          return;
             end if;
end loop;
raise Item_Is_Not_In_Set;
      procedure Union (Of_The_Set : in Set;
And_The_Set : in Set;
To_The_Set : in out Set) is
To_Index : Natural;
To_Back : Natural;
       begin
             else
                                To_Index := To_Index - 1;
                          end if;
                    end loop;
if To_Index = 0 then
             ir To_index = 0 then
    To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
        And_The_Set.The_Items(And_Index);
        To_The_Set.The_Back := To_The_Set.The_Back + 1;
end if;
end loop;
sption
       exception
when Constraint_Error =>
raise Overflow;
       end Union:
      procedure Intersection (Of_The_Set : in Set;
And_The_Set : in Set;
To_The_Set : in out Set) is
```

```
And_Index : Natural;
      Or_ine_set.rne_items(or_index) =
And_The_Set.The_Items(And_Index) then
To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
Of_The_Set.The_Items(Of_Index);
To_The_Set.The_Back := To_The_Set.The_Back + 1;
    and_Index := And_Index - 1;
end if;
end loop;
end loop;
eption
exception
when Constraint_Error =>
raise Overflow;
end Intersection;
And_Index : Natural;
begin
      To_The_Set.The_Back := 0;
      To_The_Set.The_Back := 0;
for Of_Index in 1 . . Of_The_Set.The_Back loop
And_Index := And_The_Set.The_Back;
while And_Index > 0 loop
    if Of_The_Set.The_Items(Of_Index) =
        And_The_Set.The_Items(And_Index) then
                 And_Index := And_Index - 1;
end if;
           end 1r;
end loop;
if And_Index = 0 then
   To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
    Of_The_Set.The_Items(Of_Index);
   To_The_Set.The_Back := To_The_Set.The_Back + 1;
      end if;
end loop;
exception when Constraint_Error =>
raise Overflow;
end Difference;
modified by Tuan Nguyen
20 Aug 95
replacing functions with procedures
                                           /T.oft
procedure Is_Equal
                                            Right
Result
Result := Is_Equal(Left, Right);
end Is_Equal;
procedure Extent_Of
                                           (The Set
                                                             : in Set;
                                                             : out Natural) is
begin
      Result := Extent_Of(The_Set);
end Extent_Of;
                                           (The_Set
Result
procedure Is Empty
                                                             : out Boolean) is
Result := Is_Empty(The_Set);
end Is_Empty;
                                           (The_Item : in Item;
Of_The_Set : in Set;
Result : out Boolean) is
procedure Is_A_Member
Result := Is_A_Member(The_Item,Of_The_Set);
end Is_A_Member;
                                           (Left
procedure Is_A_Subset
                                                             : in Set:
                                            Right
Result
                                                             · in Set:
                                                             : out Boolean) is
      Result := Is_A_Subset(Left, Right);
end Is_A_Subset;
procedure Is_A_Proper_Subset (Left
                                            Right
Result
                                                             : in Set;
                                                             : out Boolean) is
Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
end of modification
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

```
TYPE Set_Simple_Sequential_Bounded_Managed_Noniterator SPECIFICATION
  GENERIC
Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
     INPUT
From_The_Set : Set,
To_The_Set : Set
     To_The_Set : Set
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END
  OPERATOR Clear
SPECIFICATION
    INPUT
The_Set : Set
OUTPUT
The_Set : Set
     EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Add
SPECIFICATION
    INPUT
        The_Item : Item,
To_The_Set : Set
       To_The_Set : Set
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 OPERATOR Remove
SPECIFICATION
    INPUT
The_Item : Item,
From_The_Set : Set
OUTPUT
    From_The_Set : Set
EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 EMD
 OPERATOR Union
 SPECIFICATION
INPUT
Of_The_Set : Set,
   Of_The_Set : Set,
And_The_Set : Set,
To_The_Set : Set
OUTPUT
To_The_Set : Set
EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 END
OPERATOR Intersection SPECIFICATION
   PECIFICATION
INPUT
Of_The_Set : Set,
And_The_Set : Set,
To_The_Set : Set,
   OUTPUT
To_The_Set : Set
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
END
OPERATOR Difference
SPECIFICATION
       Of_The_Set : Set,
```

```
And_The_Set : Set,
To_The_Set : Set
OUTPUT
      OUTPUT
TO_The_Set : Set
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
    OPERATOR IS_Equal SPECIFICATION INPUT
      Left : Set,
Right : Set
OUTPUT
         Result : Boolean
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Extent_Of SPECIFICATION
      INPUT
        The_Set : Set
      OUTPUT
Result : Natural
      FYCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OFERATOR IS_Empty
SPECIFICATION
INPUT
        The_Set : Set
      OUTPUT
      Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Member SPECIFICATION
     TNPITT
        The_Item : Item,
Of_The_Set : Set
     OUTPUT
        Result : Boolean
     EXCEPTIONS
Overflow, Item_is_In_Set, Item_is_Not_In_Set
  END
  OPERATOR Is_A_Subset
SPECIFICATION
INPUT
Left : Set,
     Right : Set
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Proper_Subset
SPECIFICATION
INPUT
Left : Set,
        Right : Set
     OUTPUT
Result : Boolean
EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
IMPLEMENTATION ADA Set_Simple_Sequential_Bounded_Managed_Noniterator
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
type Item is private;
package Set_Simple_Sequential_Unbounded_Managed_Iterator is
       type Set is limited private;
                                               (From_The_Set : in out Set);
(The_Set : in out Set);
(The_Item : in Item;
To_The_Set : in out Set);
(The_Item : in Item;
(The_Item : in Item;
From_The_Set : in out Set);
(Of_The_Set : in out Set);
       procedure Copy
       procedure Clear
procedure Add
       procedure Remove
       procedure Union
       And The Set
To_The_Set
procedure Intersection (Of_The_Set
                                                                            in
                                                                                        Set:
                                                                            in
in
                                                                                out Set);
                                                And_The_Set
To_The_Set
(Of_The_Set
                                                                            in
                                                                           in out Set);
in Set;
        procedure Difference
                                                  And_The_Set
To_The_Set
                                                                            in
                                                                           in out Set);
       modified by Tuan Nguyen
        20 Aug 95 replacing functions with procedures
                                                                                  in Set;
in Set;
        procedure Is_Equal
                                                           (Left
                                                           Right
Result
(The_Set
                                                                                   in Set;
out Boolean);
in Set;
out Natural);
in Set;
out Boolean);
        procedure Extent_Of
                                                             Result
                                                           (The_Set
Result
(The_Item
        procedure Is_Empty
                                                                                   in Item;
        procedure Is_A_Member
```

```
Of_The_Set : in Set;
Result : out Boolean);
(Left : in Set;
Right : in Set;
                                                       Result
(Left
Right
     procedure Is_A_Subset
                                                                           : in Set;
: out Boolean);
: in Set;
: in Set;
: out Boolean);
                                                       Result
(Left
Right
     procedure Is_A_Proper_Subset
                                                         Result
-- end of modification
                                                                         : in Set;
: in Set)
: in Set)
: in Set)
: in Item;
: in Set)
: in Set;
                                                     (Left
Right
(The_Set
(The_Set
(The_Item
Of_The_Set
(Left
Bight
      function Is_Equal
                                                                                            return Boolean;
                                                                                            return Natural;
return Boolean;
      function Extent_Of
function Is_Empty
function Is_A_Member
                                                                                            return Boolean;
      function Is_A_Subset
                                                                                            return Boolean;
      function Is_A_Subset Right
function Is_A_Proper_Subset (Left
Right
                                                                          : in Set;
: in Set)
                                                                                           return Boolean;
      : exception;
      Overflow
      Item_Is_In_Set : exception;
Item_Is_Not_In_Set : exception;
private
    type Node;
    type Set is access Node;
end Set_Simple_Sequential_Unbounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
---
with Storage_Manager_Sequential;
package body Set_Simple_Sequential_Unbounded_Managed_Iterator is
              record
                     The_Item : Item;
Next : Set;
              end record;
       procedure Free (The_Node : in out Node) is begin
       null;
end Free;
       Set) is
       The_Node.Next := To_Next;
end Set_Next;
       function Next_Of (The_Node : in Node) return Set is
       begin
              return The Node.Next;
       end Next_Of;
       package Node_Manager is new Storage_Manager_Sequential
                                                                    Item => Node,
Pointer => Set,
Free => Free,
Set_Pointer => Set_Next,
                                                                   (Item
                                                                    Pointer_Of => Next_Of);
      in Node_Manager.Free(To_The_Set);
if From_The_Set /= null then
    To_The_Set := Node_Manager.New_Item;
    To_The_Set.The_Item := From_Index.The_Item;
    To_Index := To_The_Set;
    From_Index := To_The_Set;
    From_Index := To_Index.Next;
    while From_Index /= null loop
    To_Index.Next := Node_Manager.New_Item;
    To_Index.The_Item := From_Index.The_Item;
    To_Index.The_Item := From_Index.The_Item;
    From_Index := From_Index.Next;
    end loop;
       begin
              end loop;
end if;
       exception
when Storage_Error =>
raise Overflow;
       end Copy:
       procedure Clear (The_Set : in out Set) is begin
              Node_Manager.Free(The_Set);
      begin
              while Index /= null loop
   if Index.The_Item = The_Item then
     raise Item_Is_In_Set;
              else
Index := Index.Next;
end if;
end loop;
Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Item := The_Item;
Temporary_Node.Next := To_The_Set;
To_The_Set := Temporary_Node;
       exception
       when Storage_Error =>
    raise Overflow;
end Add;
```

```
: Set := From_The_Set;
       in
while Index /= null loop
   if Index.The_Item = The_Item then
    if Previous = null then
        From_The_Set := From_The_Set.Next;
                      else
                      Previous.Next := Index.Next;
end if;
Index.Next := null;
                      Node_Manager.Free(Index);
                      return;
               else
               Previous := Index;
Index := Index.Next;
end if;
end loop;
end loop;
raise Item_Is_Not_In_Set;
end Remove;
To_Index
To_Top
       Temporary_Node : Set;
      To_Index := To_Index.Next;
end if;
              end loop;
if To_Index = null then
              11 10_LINGEX = NUII THEN
TEMPORARY_NOde := Node_Manager.New_Item;
Temporary_Node.The_Item := From_Index.The_Item;
Temporary_Node.Next := To_The_Set;
To_The_Set := Temporary_Node;
end if;
       From_Index := From_Index.Next;
end loop;
exception
      ption
when Storage_Error =>
    raise Overflow;
procedure Intersection (Of_The_Set
      And The Set : in Set;
And The Set : in Set;
To_The Set : in out Set) is
Of_Index : Set := Of_The_Set;
And_Index : Set;
Temporary_Node : Set;
in
      in
Node_Manager.Free(To_The_Set);
while Of_Index /= null loop
And_Index := And_The_Set;
while And_Index /= null loop
    if Of_Index.The_Item = And_Index.The_Item then
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := Of_Index.The_Item;
        Temporary_Node.Next := To_The_Set;
        To_The_Set := Temporary_Node;
        exit;
      else
And_Index := And_Index.Next;
end if;
end loop;
of_Index := Of_Index.Next;
end loop;
eption
exception
       when Storage Error =>
end Intersection;
procedure Difference (Of_The_Set : in
                                   And_The_Set : in Set;
To_The_Set : in out Set) is
: Set := Of_The_Set;
       Of Index
       And_Index : Set;
Temporary_Node : Set;
begin
       Node_Manager.Free(To_The_Set);
while Of_Index /= null loop
   And_Index := And_The_Set;
```

```
while And_Index /= null loop
   if Of_Index.The_Item = And_Index.The_Item then
      exit;
                 else
                 And_Index := And_Index.Next;
end if;
            end loop;
if And_Index = null then
                 and_Index = null then
Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Item := Of_Index.The_Item;
Temporary_Node.Next := To_The_Set;
To_The_Set := Temporary_Node;
            end if:
     Of_Index := Of_Index.Next;
end loop;
exception
when Storage_Error =>
raise Overflow;
end Difference;
modified by Tuan Nguyen
20 Aug 95
replacing functions with procedures
                                          meft
                                                          : in Set:
procedure Is_Equal
                                                          : in Set;
: out Boolean) is
                                           Right
                                           Result
 begin
      Result := Is_Equal(Left, Right);
 end Is_Equal;
                                         (The Set
procedure Extent_Of
                                                          : out Natural) is
                                           Result
      Result := Extent_Of(The_Set);
 end Extent Of:
                                                         : in Set;
 procedure Is_Empty
                                          (The_Set
                                                          : out Boolean) is
 Result := Is_Empty(The_Set);
end Is_Empty;
                                         (The_Item : in Item;
Of_The_Set : in Set;
Result : out Bool
 procedure Is_A_Member
                                                          : out Boolean) is
 Result := Is_A_Member(The_Item,Of_The_Set);
end Is_A_Member;
                                                          : in Set;
: in Set;
: out Boolean) is
                                          (Left
 procedure Is_A_Subset
                                           Right
                                           Result
 begin
      Result := Is_A_Subset(Left, Right);
 end Is_A_Subset;
 procedure Is_A_Proper_Subset (Left
                                                          : in Set;
                                                          : in Set;
: out Boolean) is
                                          Right
Result
 Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
 end of modification
 else
                       Right_Index := Right_Index.Next;
            end if;
end loop;
if Right_Index = null then
                  return False;
             Left_Count := Left_Count + 1;
Left_Index := Left_Index.Next;
end if;
       end if;
end loop;
Right_Index := Right;
while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
end loop;
return (Left_Count = Right_Count);
Tableson;
```

end Is_Equal;

```
function Extent_Of (The_Set : in Set) return Natural is
         Count : Natural := 0;
Index : Set := The_Set;
   begin
         while Index /= null loop
              Count := Count + 1;
Index := Index.Next;
         end loop;
return Count;
    end Extent Of:
    function Is_Empty (The_Set : in Set) return Boolean is
    return (The_Set = null);
end Is_Empty;
    function Is_A_Member (The_Item : in Item;
Of_The_Set : in Set) return Boolean is
         Index : Set := Of_The_Set;
    begin
         in
while Index /= null loop
   if The_Item = Index.The_Item then
        return True;
end if;
   Index := Index.Next;
         end loop;
return False;
    end Is_A_Member;
   begin
         while Left_Index /= null loop
               Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
                         exit;
                    Right_Index := Right_Index.Next;
end if;
               end loop;
if Right_Index = null then
return False;
               else
         else
Left_Index := Left_Index.Next;
end if;
end loop;
return True;
    end Is_A_Subset;
    Right_Index := Right_Index.Next;
                     end if;
               end loop;
if Right_Index = null then
False;
               else
               Left_Count := Left_Count + 1;
Left_Index := Left_Index.Next;
end if;
         end 1r;
end loop;
Right_Index := Right;
while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
end loop;
return (Left_Count < Right_Count);
It & Proper Subset:
     end Is_A_Proper_Subset;
    begin
  while The_Iterator /= null loop
    Process(The_Iterator.The_Item, Continue);
    exit when not Continue;
    The_Iterator := The_Iterator.Next;
    end loop;
end Iterate;
end Set_Simple_Sequential_Unbounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
TYPE Set_Simple_Sequential_Unbounded_Managed_Iterator SPECIFICATION
  GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
     INPUT
From_The_Set : Set,
To_The_Set : Set
     OUTPUT
To_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Clear
   SPECIFICATION
INPUT
The_Set : Set
      OUTPUT
     The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Add
  OPERATOR AGG
SPECIFICATION
INPUT
The_Item : Item,
To_The_Set : Set
      OUTPUT
     To_The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   EMD
  OPERATOR Remove
SPECIFICATION
     INPUT
The_Item : Item,
From_The_Set : Set
OUTPUT
     OUTPUT
From_The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   ENO
  OPERATOR Union
SPECIFICATION
     INPUT
Of The Set : Set,
And The Set : Set
To The Set : Set
OUTPUT
To The Set : Set
EXCEPTIONS
OUTPUT
TO THE SET : Set
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Intersection
  SPECIFICATION
INPUT
Of_The_Set : Set,
     And_The_Set : Set,
To_The_Set : Set
OUTPUT
To_The_Set : Set
      EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Difference
   SPECIFICATION
INPUT
Of_The_Set : Set,
     And_The_Set : Set,
To_The_Set : Set
OUTPUT
To_The_Set : Set
      EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
```

```
END
  OPERATOR Is_Equal SPECIFICATION
    INPUT
Left : Set,
Right : Set
    OUTPUT
Result : Boolean
EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END
  OPERATOR Extent_Of
  SPECIFICATION
INPUT
The_Set : Set
     OUTPUT
    Result : Natural
EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_Empty
SPECIFICATION
    INPUT
The_Set : Set
OUTPUT
     Result : Boolean
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Member
SPECIFICATION
INPUT
     The_Item : Item,
Of_The_Set : Set
OUTPUT
       Result : Boolean
     EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Subset SPECIFICATION
    INPUT
Left : Set,
Right : Set
     OUTPUT
    Result : Boolean
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Proper_Subset SPECIFICATION
    INPUT
Left : Set,
Right : Set
OUTPUT
       Result : Boolean
    EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Iterate
SPECIFICATION
GENERIC
       Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
     INPUT
       Over_The_Set : Set
     EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
IMPLEMENTATION ADA Set_Simple_Sequential_Unbounded_Managed_Iterator
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
type Item is private;
package Set_Simple_Sequential_Unbounded_Managed_Noniterator is
      type Set is limited private;
                                     procedure Copy
     procedure Clear
procedure Add
     procedure Remove
     procedure Union
                                     And_The_Set
To_The_Set
(Of_The_Set
And_The_Set
To_The_Set
(Of_The_Set
And_The_Set
To_The_Set
                                                         : in Set;
: in out Set);
: in Set;
     procedure Intersection
     procedure Difference
                                                         : in out Set);
     modified by Tuan Nguyen
     20 Aug 95
replacing functions with procedures
     procedure Is_Equal
                                              (Left
                                                               : in Set:
                                               Right
Result
                                                               : in Set;
: out Boolean);
     procedure Extent_Of
                                               (The Set
                                                               : in Set:
                                                               : out Natural);
```

```
: in Set;
: out Boolean);
: in Item;
                                                (The_Set
Result
     procedure Is_Empty
     procedure Is_A_Member
                                                 (The Item
                                                  Of_The_Set : in Set;
Result : out Boo
                                                                     out Boolean);
in Set;
     procedure Is_A_Subset
                                                 Left
                                                 Right
Result
                                                                     in Set
                                                                  : out Boolean);
: in Set;
     procedure Is_A_Proper_Subset
                                                (Left
                                                 Right
Result
                                                                  : in Set:
-- end of modification
                                                                : in Set;
: in Set)
: in Set)
: in Set)
                                               (Left
     function Is Equal
                                               (Left
Right
(The_Set
(The_Set
(The_Item
Of_The_Set
(Left
                                                                                return Boolean:
     function Extent_Of
                                                                                 return Natural:
                                                                                 return Boolean;
     function Is_Empty
function Is_A_Member
                                                                   in Item;
in Set)
in Set;
in Set;
                                                                                return Boolean:
     function Is_A_Subset
                                                                                return Boolean;
                                                Right
     function Is_A_Proper_Subset (Left
                                                                   in Set:
                                                Right
                                                                                return Boolean;
     Overflow : exception; Item_Is_In_Set : exception; Item_Is_Not_In_Set : exception;
private
type Node;
type Set is access Node;
end Set_Simple_Sequential_Unbounded_Managed_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                      "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body Set_Simple_Sequential_Unbounded_Managed_Noniterator is
       type Node is
              record
The_Item : Item;
                     Next
                                    : Set;
              end record:
       procedure Free (The_Node : in out Node) is begin
       null;
end Free;
      Set) is
       begin
              The Node.Next := To_Next;
       end Set_Next;
       function Next_Of (The_Node : in Node) return Set is
       begin
              return The_Node.Next;
       end Next_Of;
       package Node_Manager is new Storage_Manager_Sequential
                                                                (Item
                                                                                      => Node,
                                                                 Pointer => Node,
Free => Set,
Set_Pointer => Set_Next,
                                                                 Pointer_Of => Next_Of);
      procedure Copy (From_The_Set : in Set;

The Set : in out Set) is
              To_The_Set : in out
From_Index : Set := From_The_Set;
To_Index : Set;
       begin
             in
Node_Manager.Free(To_The_Set);
if From_The_Set /= null then
To_The_Set := Node_Manager.New_Item;
To_The_Set.The_Item := From_Index.The_Item;
To_Index := To_The_Set;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := Node_Manager.New_Item;
To_Index := To_Index.Next;
To_Index.The_Item := From_Index.The_Item;
From_Index := From_Index.Next;
end loop;
              end loop;
end if;
       end 11;
exception
when Storage_Error =>
raise Overflow;
       procedure Clear (The_Set : in out Set) is begin
       Node_Manager.Free(The_Set);
end Clear;
       begin
              nwhile Index /= null loop
   if Index.The_Item = The_Item then
      raise Item_Is_In_Set;
                     else
                            Index := Index.Next;
                     end if;
              end loop;
              end loop;
Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Item := The_Item;
Temporary_Node.Next := To_The_Set;
To_The_Set := Temporary_Node;
       exception
when Storage_Error =>
raise Overflow;
       end Add;
```

```
: Set := From_The_Set;
begin
        while Index /= null loop
if Index.The_Item = The_Item then
if Previous = null then
From_The_Set := From_The_Set.Next;
                           else
                          Previous.Next := Index.Next;
end if;
Index.Next := null;
                          Node_Manager.Free(Index);
                 Previous := Index;
Index := Index.Next;
end if;
                 else
end lif;
end loop;
raise Item_Is_Not_In_Set;
end Remove;
Temporary_Node : Set;
        in
Node_Manager.Free(To_The_Set);
while From_Index /= null loop
   Temporary_Node := Node_Manager.New_Item;
   Temporary_Node.The_Item := From_Index.The_Item;
   Temporary_Node.Next := To_The_Set;
   To_The_Set := Temporary_Node;
   From_Index := From_Index.Next;
         end loop;
From_Index := And_The_Set;
To_Top := To_The_Set;
while From_Index /= null loop
                 exit;
else
   To_Index := To_Index.Next;
end if;
                 end 1t;
end loop;
if To_Index = null then
  Temporary_Node := Node_Manager.New_Item;
  Temporary_Node.The_Item := From_Index.The_Item;
  Temporary_Node.Next := To_The_Set;
  To_The_Set := Temporary_Node;
end if;
From_Index := From_Index_Next.
                  From Index := From Index.Next:
         end loop;
exception
when Storage_Error =>
                 raise Overflow;
procedure Intersection (Of_The_Set : in Set;
And_The_Set : in Set;
To_The_Set : in out Set) is
Of_Index : Set := Of_The_Set;
         And_Index : Set;
Temporary_Node : Set;
        in
  Node_Manager.Free(To_The_Set);
while Of_Index /= mull loop
  And_Index := And_The_Set;
  while And_Index /= null loop
    if Of_Index.The_Item = And_Index.The_Item then
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := Of_Index.The_Item;
        Temporary_Node.Next := To_The_Set;
        To_The_Set := Temporary_Node;
        exit:
                                   exit;
                           else
                  And_Index := And_Index.Next;
end if;
end loop;
Of_Index := Of_Index.Next;
          end loop;
 exception
when Storage_Error =>
 raise Overflow;
end Intersection;
                                          ce (Of_The_Set : in
And_The_Set : in
To_The_Set : in
: Set := Of_The_Set;
procedure Difference (Of_The_Set
                                                                          : in out Set) is
          And Index
                                          : Set:
          Temporary_Node : Set;
 begin
          n
Node_Manager.Free(To_The_Set);
while Of_Index /= null loop
And_Index := And_The_Set;
```

```
while And_Index /= null loop
   if Of_Index.The_Item = And_Index.The_Item then
                           exit;
                      else
                     And_Index := And_Index.Next;
end if;
               end ir;
end loop;
if And_Index = null then
   Temporary_Node := Node_Manager.New_Item;
   Temporary_Node.The_Item := Of_Index.The_Item;
   Temporary_Node.Next := To_The_Set;
   To_The_Set := Temporary_Node;
                end if;
Of_Index := Of_Index.Next;
          end loop;
     exception
when Storage_Error =>
     raise Overflow;
end Difference;
    modified by Tuan Nguyen
    20 Aug 95
replacing functions with procedures
                                                             : in Set:
                                             (Left
     procedure Is_Equal
                                              Right
Result
                                                             · in Set:
                                                             : out Boolean) is
     Result := Is_Equal(Left, Right);
end Is_Equal;
                                             (The_Set
Result
                                                           : in Set;
: out Natural) is
          Result := Extent_Of(The_Set);
     end Extent Of;
                                             (The_Set
Result
                                                          : in Set;
: out Boolean) is
    procedure Is_Empty
          Result := Is_Empty(The_Set);
    end Is Empty:
                                             (The_Item : in Item;
Of_The_Set : in Set;
Result : out Boolean) is
    procedure Is_A_Member
          Result := Is_A_Member(The_Item,Of_The_Set);
     end Is A Member:
    procedure Is_A_Subset
                                             /Left
                                                            · in Set:
                                              Result
    Result := Is_A_Subset(Left,Right);
end Is_A_Subset;
    procedure Is_A_Proper_Subset (Left
                                                            : in Set:
                                                            : in Set;
: out Boolean) is
                                              Right
Result
    Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
-- end of modification
    while Left_Index /= null loop
                Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
exit;
                     else
                    Right_Index := Right_Index.Next;
end if;
                end loop;
if Right_Index = null then
return False;
                else
    Left_Count := Left_Count + 1;
    Left_Index := Left_Index.Next;
end if;
          end loop:
          Right_Index := Right;
```

```
while Right_Index /= null loop
  Right_Count := Right_Count + 1;
  Right_Index := Right_Index.Next;
              end loop;
return (Left_Count = Right_Count);
       end Is_Equal;
       function Extent_Of (The_Set : in Set) return Natural is
              Count : Natural := 0;
Index : Set := The_Set;
       begin
              while Index /= null loop
Count := Count + 1;
Index := Index.Next;
              end loop;
return Count;
       end Extent Of:
       function Is_Empty (The_Set : in Set) return Boolean is
              return (The_Set = null);
       end Is_Empty;
       function Is_A_Member (The_Item : in Item;
Of_The_Set : in Set) return Boolean is
              Index : Set := Of_The_Set;
       begin
             in
while Index /= null loop
   if The_Item = Index.The_Item then
        return True;
   end if;
   Index := Index.Next;
              end loop;
return False;
       end Is_A_Member;
      begin
while Left_Index /= null loop
                     Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
exit;
                            else
                     Right_Index := Right_Index.Next;
end if;
end loop;
if Right_Index = null then
              if Right_Index = null then
    return False;
else
        Left_Index := Left_Index.Next;
end if;
end loop;
return True;
       end Is_A_Subset;
      function Is_A_Proper_Subset (Left : in Set;
    Right : in Set) return Boolean is
    Left_Count : Natural := 0;
    Right_Count : Natural := 0;
    Left_Index : Set := Left;
    Right_Index : Set;
begin
      Right_Index . ----
begin
while Left_Index /= null loop
Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
exit;
else
                     esse
Right_Index := Right_Index.Next;
end if;
end loop;
if Right_Index = null then
return False;
                     Left_Count := Left_Count + 1;
    Left_Index := Left_Index.Next;
end if;
             end 11,
end loop;
Right_Index := Right;
while Right_Index /= mull loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
       end loop;
  return (Left_Count < Right_Count);
end Is_A_Proper_Subset;</pre>
end Set_Simple_Sequential_Unbounded_Managed_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
TYPE Set_Simple_Sequential_Unbounded_Managed_Noniterator SPECIFICATION GENERIC
     Item : PRIVATE_TYPE
  Item : PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_Set : Set,
To_The_Set : Set
OUTPUT
To_The_Set : Set
EXCEPTIONS
OUTPIONS
OUTPIONS
OUTPIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   END
   OPERATOR Clear
  SPECIFICATION
INPUT
         The_Set : Set
      OUTPUT
     The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   DATE:
   OPERATOR Add
   SPECIFICATION
     INPUT
The_Item : Item,
To_The_Set : Set
      OUTPUT
To_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Remove
   SPECIFICATION
INPUT
The_Item : Item,
         From_The_Set : Set
      OUTPUT
From_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   EMD
   OPERATOR Union SPECIFICATION
      INPOT
Of The Set : Set,
And The Set : Set,
To The Set : Set
      OUTPUT
To_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   מעפ
   OPERATOR Intersection
SPECIFICATION
INPUT
Of_The_Set : Set,
And_The_Set : Set,
To_The_Set : Set
      OUTPUT
To_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   END
   OPERATOR Difference
    SPECIFICATION
      INPUT
Of_The_Set : Set,
```

```
And_The_Set : Set,
To_The_Set : Set
OUTPUT
To_The_Set : Set
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_Equal
SPECIFICATION
INPUT
Left : Set,
Right : Set
OUTPUT
RESULT : Boolean
EXCEPTIONS
     EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Extent_Of
SPECIFICATION
INPUT
        The_Set : Set
     OUTPUT
Result : Natural
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Set : Set
     OUTPUT
     Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Member SPECIFICATION
     INPUT
The_Item : Item,
Of_The_Set : Set
     OUTPUT
        Result : Boolean
     EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END
  OPERATOR Is_A_Subset
SPECIFICATION
     INPUT
Left : Set,
Right : Set
     OUTPUT
     Result : Boolean
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is A Proper_Subset
SPECIFICATION
INPUT
Left : Set,
Right : Set
OUTPUT
        Result : Boolean
     EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
IMPLEMENTATION ADA Set_Simple_Sequential_Unbounded_Managed_Noniterator
END
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
type Item is private;
package Set_Simple_Sequential_Unbounded_Unmanaged_Iterator is
      type Set is limited private;
                                       procedure Copy
     procedure Clear
procedure Add
     procedure Remove
                                       (The_Item : in out Set);
From_The_Set : in out Set);
(Of_The_Set : in Set;
And_The_Set : in Set;
     procedure Union
                                                           : in out Set);
; in Set:
                                      And_The_Set
To_The_Set
(Of_The_Set
And_The_Set
To_The_Set
(Of_The_Set
And_The_Set
To_The_Set
     procedure Intersection
                                                             in Set;
in out Set);
     procedure Difference
                                                           : in
                                                                       Set;
                                                          : in Set;
: in out Set);
     modified by Tuan Nguyen
     20 Aug 95
replacing functions with procedures
                                               (Left
                                                                 : in Set;
     procedure Is_Equal
                                                 Right
Result
                                                                 : in Set;
: out Boolean);
: in Set;
     procedure Extent_Of
                                                (The Set
                                               Result
(The_Set
Result
                                                                   out Natural);
                                                                : in Set;
: out Boolean);
     procedure Is_Empty
     procedure Is_A_Member
                                               (The_Item
                                                                : in Item:
```

```
Of_The_Set : in Set;
Result : out Boolean);
(Left : in Set;
     procedure Is_A_Subset
                                           Right
Result
     procedure Is_A_Proper_Subset (Left
                                                          : in Set:
                                           Result
  - end of modification
                                         (Left
                                                        : in Set;
     function Is Equal
                                         Right
(The_Set
(The_Set
(The_Item
Of_The_Set
                                                        : in Set)
: in Set)
: in Set)
: in Item;
                                                                      return Boolean;
     function Extent_Of
                                                                      return Natural;
return Boolean;
     function Is_Empty
function Is_A_Member
                                                                      return Boolean:
                                                           in Set)
                                         (Left
Right
                                                           in Set;
in Set)
     function Is_A_Subset
                                                                     return Boolean;
     function Is_A_Proper_Subset (Left
                                                           in Set:
                                                                     return Boolean;
    procedure Iterate (Over_The_Set : in Set);
     Overflow
                             : exception:
     Item_Is_In_Set : exception;
Item_Is_Not_In_Set : exception;
type Node;
  type Set is access Node;
end Set_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of PAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body Set_Simple_Sequential_Unbounded_Unmanaged_Iterator is
     type Node is
         record
              The_Item : Item;
Next : Set;
          end record;
    begin
if From_The_Set = null then
The Set := null;
              From_Index.The_Item,
                                                               => null):
                   To_Index := To_Index.Next;
From_Index := From_Index.Next;
         end loop;
end if;
     exception
when Storage_Error =>
              raise Overflow:
    procedure Clear (The_Set : in out Set) is begin
     The_Set := null;
end Clear;
    in while Index /= null loop while Index /= null loop Them = The_Item then
     begin
         when Storage_Error =>
raise Overflow;
end Add;
     procedure Remove (The_Item : in Item; From_The_Set : in out Set) is
          From_The_Set : in
Previous : Set;
Index : Set := From_The_Set;
     begin
          Previous.Next := Index.Next;
end if;
                    return;
         Previous := Index;
Index := Index.Next;
end if;
end loop;
          raise Item_Is_Not_In_Set;
     end Remove;
```

```
To_The_Set := null;
while From_Index /= null loop
To_The_Set := new Node'(The_Item => From_Index.The_Item,
Next => To_The_Set);
                       From_Index := From_Index.Next;
              end loop;
From_Index := And_The_Set;
To_Top := To_The_Set;
while From_Index /= null loop
                      it from Index := To_Top;
to_Index := To_Top;
while To_Index /= null loop
    if From_Index.The_Item = To_Index.The_Item then
        exit;
                      exit;
else
To_Index := To_Index.Next;
end if;
end loop;
if To_Index = null then
To_The_Set := new Node'(The_Item =>
From Index. The Item,
                                                                          Next
                                                                                           => To_The_Set);
                      From_Index := From_Index.Next;
              end loop;
       exception
when Storage_Error =>
raise Overflow;
       end Union:
      procedure Intersection (Of_The_Set : in Set;
And_The_Set : in Set;
To_The_Set : in out Set) is
Of_Index : Set := Of_The_Set;
And_Index : Set;
              To_The_Set := null;
while Of_Index /= null loop
And_Index := And_The_Set;
while And_Index /= null loop
    if Of_Index.The_Item = And_Index.The_Item then
        To_The_Set := new Node'(The_Item =>
       begin
Of_Index.The_Item,
                                                                                                   => To_The_Set);
             else
And_Index := And_Index.Next;
end if;
end loop;
of_Index := Of_Index.Next;
end loop;
end loop;
eption
       exception
       when Storage_Error =>
raise Overflow;
end Intersection;
       procedure Difference (Of_The_Set : in Set;
And_The_Set : in Set;
To_The_Set : in out Set) is
And_Index : Set := Of_The_Set;
And_Index : Set;
               To_The_Set:= null;
              ro_rne_set:= mult;
while Of_Index /= mull loop
And_Index := And_The_set;
while And_Index /= mull loop
if Of_Index.The_Item = And_Index.The_Item then
                                    exit:
                              else
                              And_Index := And_Index.Next;
end if;
                      end loop;
if And_Index = null then
    To_The_Set := new Node'(The_Item => Of_Index.The_Item,
    Next => To_The_Set);
                      end if;
Of_Index := Of_Index.Next;
               end loop;
       exception when Storage_Error =>
       raise Overflow;
end Difference;
       modified by Tuan Nguyen
-- 20 Aug 95
-- replacing functions with procedures
       procedure Is_Equal
                                                              (Left
                                                                                     : in Set:
                                                                Right
Result
                                                                                     : in Set;
: out Boolean) is
       begin
               Result := Is_Equal(Left, Right);
        end Is_Equal;
                                                              (The_Set
Result
                                                                                     : in Set;
: out Natural) is
       procedure Extent Of
```

Result := Extent_Of(The_Set);

```
end Extent Of:
                                                      : in Set;
procedure Is_Empty
                                        (The Set
                                                        : out Boolean) is
      Result := Is_Empty(The_Set);
end Is_Empty;
                                         (The_Item : in Item;
Of_The_Set : in Set;
Result : out Boolean) is
                                        (The Item
procedure Is_A_Member
      Result := Is_A_Member(The_Item,Of_The_Set);
end Is_A_Member;
                                        (Left
                                                        · in Set:
procedure Is_A_Subset
                                                        : in Set;
: out Boolean) is
                                         Right
Result
      Result := Is_A_Subset(Left,Right);
end Is_A_Subset;
                                                        : in Set;
: in Set;
: out Boolean) is
procedure Is_A_Proper_Subset (Left
                                         Right
Result
Degin
    Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
end of modification
begin
     while Left_Index /= null loop
   Right_Index := Right;
   while Right_Index /= null loop
    if Left_Index.The_Item = Right_Index.The_Item then
                      exit:
               else
   Right_Index := Right_Index.Next;
end if;
           end loop;
if Right_Index = null then
return False;
           Left_Count := Left_Count + 1;
    Left_Index := Left_Index.Next;
end if;
     end lr;
end loop;
Right_Index := Right;
while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
end loop;
return (Left_Count = Right_Count);
end Is_Equal;
while Index /= null loop
Count := Count + 1;
Index := Index.Next;
end loop;
return Count;
end Extent_Of;
 function Is_Empty (The_Set : in Set) return Boolean is
begin
     return (The_Set = null);
 end Is_Empty;
```

function Is_A_Member (The_Item : in Item;

```
Of_The_Set : in Set) return Boolean is
Index : Set := Of_The_Set;
     begin while Index /= null loop if The_Item = Index.The_Item then True:
                   return True;
end if;
Index := Index.Next;
      end loop;
return False;
end Is_A_Member;
     begin
            while Left_Index /= null loop
Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
                                exit:
                         exit;
else
   Right_Index := Right_Index.Next;
end if;
                   end loop;
if Right_Index = null then
return False;
                   else
    Left_Index := Left_Index.Next;
end if;
      end loop;
return True;
end Is_A_Subset;
     begin
            while Left_Index /= null loop
   Right_Index := Right;
   while Right_Index /= null loop
    if Left_Index.The_Item = Right_Index.The_Item then
                                exit:
                         Right_Index := Right_Index.Next;
end if;
                   end loop;
if Right_Index = null then
return False;
                   else
                   Left_Count := Left_Count + 1;
Left_Index := Left_Index.Next;
end if;
            end 1r;
end loop;
Right_Index := Right;
while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
      end loop;
return (Left_Count < Right_Count);
end Is_A_Proper_Subset;
      procedure Iterate (Over_The_Set : in Set) is
    The_Iterator : Set := Over_The_Set;
    Continue : Boolean;
      begin
            in
while The_Iterator /= null loop
    Process(The_Iterator.The_Item, Continue);
    exit when not Continue;
    The_Iterator := The_Iterator.Next;
end Set_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
TYPE Set Simple_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_Set: Set,
        To_The_Set : Set
     OUTPUT
To_The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Clear
  SPECIFICATION
INPUT
        The_Set : Set
     OUTPUT
The_Set : Set
     EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Add
SPECIFICATION
     The_Item : Item,
To_The_Set : Set
OUTPUT
To_The_Set : Set
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Remove
   SPECIFICATION
INPUT
The_Item : Item,
     From_The_Set : Set
OUTPUT
        From_The_Set : Set
     EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Union
  OPERATOR Union
SPECIFICATION
INPUT
Of The_Set : Set,
And_The_Set : Set
To_The_Set : Set
OUTPUT
To_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Intersection
   SPECIFICATION
INPUT
Of_The_Set : Set,
      And_The_Set : Set,
To_The_Set : Set
OUTPUT
        To The Set : Set
      EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Difference SPECIFICATION
     INPUT
Of_The_Set : Set,
     Or_the_Set : Set,
And_The_Set : Set,
To_The_Set : Set
OUTPUT
To_The_Set : Set
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
```

```
END
 OPERATOR Is_Equal
 OPERATOR IS_EQU:
SPECIFICATION
INPUT
Left: Set,
Right: Set
OUTPUT
    Result : Boolean
EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 OPERATOR Extent_Of
  SPECIFICATION
INPUT
      The Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 OPERATOR IS_Empty
SPECIFICATION
    The_Set : Set
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 OPERATOR Is_A_Member SPECIFICATION
    INPUT
The_Item : Item,
Of_The_Set : Set
    OUTPUT
    Result : Boolean EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 END
 OPERATOR Is_A_Subset
SPECIFICATION
    INPUT
Left : Set,
Right : Set
    OUTPUT
    Result : Boolean
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 OPERATOR Is_A_Proper_Subset SPECIFICATION
    INPUT
Left : Set,
Right : Set
       Result : Boolean
    EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 END
 OPERATOR Iterate
SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
    INPUT
      Over_The_Set : Set
    EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
IMPLEMENTATION ADA Set_Simple_Sequential_Unbounded_Unmanaged_Iterator
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
type Item is private;
package Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator is
                      type Set is limited private;
                                                                                                                                                        procedure Copy
                    procedure Clear
procedure Add
                    procedure Remove
                 procedure Union (Of_The_Set And_The_Set To_The_Set Cof_The_Set Cof
                                                                                                                                                                                                                                       : in Set;
                                                                                                                                                         And_The_Set
To_The_Set
(Of_The_Set
                    procedure Difference
                                                                                                                                                            And_The_Set
To_The_Set
                   modified by Tuan Nguyen
                    20 Aug 95
replacing functions with procedures
                                                                                                                                                                                                                                                             : in Set;
: in Set;
: out Boolean);
: in Set;
                                                                                                                                                                                           (Left
                      procedure Is_Equal
                                                                                                                                                                                                   Result
                     procedure Extent_Of
                                                                                                                                                                                           (The Set
                                                                                                                                                                                                                                                               : out Natural);
                                                                                                                                                                                                Result
                                                                                                                                                                                                                                                                : in Set;
                    procedure Is Empty
```

```
Result :
(The_Item :
Of_The_Set :
Result :
(Left :
                                                                  : out Boolean);
: in Item;
: in Set;
     procedure Is_A_Member
                                                                     out Boolean);
in Set;
     procedure Is_A_Subset
                                                                    in Set;
                                                  Right
                                                  Result
                                                                    out Boolean);
                                                (Left
Right
     procedure Is_A_Proper_Subset
                                                  Result
                                                                  : out Boolean);
-- end of modification
     function Is_Equal
                                               (Left
                                                                 : in Set:
                                                                 : in Set;
: in Set)
: in Set)
: in Set)
: in Item;
: in Set)
                                               Right
(The_Set
(The_Set
                                                                                return Boolean;
                                                                                return Natural;
return Boolean;
     function Extent_Of
     function Is_Empty
function Is_A_Member
                                               (The_Item
Of_The_Set
(Left
                                                                                return Boolean;
     function Is_A_Subset
                                                                   in Set;
     function Is_A_Proper_Subset (Left
                                                                   in Set)
                                                                                return Boolean;
                                                                : in Set;
: in Set)
                                                                               return Boolean;
                                                Right
                                 : exception:
     Overflow
     Item_Is_In_Set : exception;
Item_Is_Not_In_Set : exception;
private
type Node;
   type Set is access Node;
end Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch -- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"

- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator is
      type Node is
          record
The_Item : Item;
               Next
                          : Set;
           end record;
     begin
    if From_The_Set = null then
        To_The_Set := null;
               To_Index := To_The_Set;
From_Index := To_The_Set;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := new Node'(The_Item => From_Index.The_Item,
                                                                   => null);
                    To_Index := To_Index.Next;
From_Index := From_Index.Next;
          end loop;
end if;
     exception
when Storage_Error =>
raise Overflow;
      end Conv:
      procedure Clear (The_Set : in out Set) is
      begin
          The_Set := null;
      end Clear;
     in
while Index /= null loop
while Index /= null loop
The Them = The_Item then
      begin
               if Index.The_Item = The_I
    raise Item_Is_In_Set;
else
          Index := Index.Next;
end if;
end loop;
To The Control
          when Storage_Error => raise Overflow;
      end Add:
     begin
          else
                     Previous.Next := Index.Next; end if;
                     return;
          else
           raise Item_Is_Not_In_Set;
      end Remove;
      procedure Union (Of_The_Set : in
           And The_Set: in Set;
To_The_Set: in out Set) is
From_Index: Set:= Of_The_Set;
           To_Index : Set;
To_Top : Set;
           To_Top
      begin
```

```
else
   To_Index := To_Index.Next;
end if;
             end loop;
if To_Index = null then
                 To_The_Set := new Node'(The_Item =>
From Index.The_Item,
                                                      => To_The_Set);
             end if:
             From_Index := From_Index.Next;
        end loop;
    exception
when Storage_Error =>
raise Overflow;
    end Union:
    And_Index .

begin

To_The_Set := null;

while Of_Index /= null loop

And_Index := And_The_Set;

while And_Index /= null loop

if Of_Index.The_Item = And_Index.The_Item then

To_The_Set := new Node'(The_Item => Next => To_The_
Of_Index.The_Item,
                                                           => To The Set):
                      exit;
                 else
             And_Index := And_Index.Next;
end if;
end loop;
             Of_Index := Of_Index.Next;
         end loop;
    exception
when Storage_Error =>
raise Overflow;
end Intersection;
    And_Index : Set;
    begin
         To_The_Set:= null;
        To_rne_set:= null;
while Of_Index /= null loop
And_Index := And_The_set;
while And_Index /= null loop
   if Of_Index.The_Item = And_Index.The_Item then
                      exit:
                 else
                 And_Index := And_Index.Next;
end if;
             end loop;
if And_Index = null then
                 end if:
         Of_Index := Of_Index.Next;
end loop;
    exception when Storage_Error =>
     raise Overflow;
end Difference;
    modified by Tuan Nguyen
    20 Aug 95
replacing functions with procedures
    procedure Is_Equal
                                     (Left
                                                   : in Set;
                                      Right
Result
                                                   : in Set:
                                                   : out Boolean) is
    begin
         Result := Is_Equal(Left, Right);
     end Is_Equal;
                                     (The_Set
Result
                                                   : in Set;
    procedure Extent Of
                                                   : out Natural) is
    begin
         Result := Extent_Of(The_Set);
```

```
end Extent Of:
     procedure Is_Empty
                                                      (The Set
                                                                        : in Set;
                                                                          : out Boolean) is
     Result := Is_Empty(The_Set);
end Is_Empty;
                                                      (The_Item : in Item;
Of_The_Set : in Set;
Result : out Boolean) is
      procedure Is_A_Member
     begin
    Result := Is_A_Member(The_Item,Of_The_Set);
end Is_A_Member;
                                                      (Left
     procedure Is_A_Subset
                                                                          · in Set:
                                                                         : in Set;
: out Boolean) is
                                                        Right
                                                       Result
     begin
            Result := Is_A_Subset(Left, Right);
      end Is A Subset;
                                                                         : in Set;
: in Set;
     procedure Is_A_Proper_Subset (Left
                                                       Right
                                                                          : out Boolean) is
                                                       Result
     begin
     Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;
-- end of modification
     Right_index . = Right loop
while Left_Index /= null loop
Right_Index := Right;
while Right_Index /= null loop
if Left_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
exit;
                        Right_Index := Right_Index.Next;
end if;
                   end loop;
if Right_Index = null then
    return False;
                   Left_Count := Left_Count + 1;
   Left_Index := Left_Index.Next;
end if;
     end if;
end loop;
Right_Index := Right;
while Right_Index /= null loop
    Right_Count := Right_Count + 1;
    Right_Index := Right_Index.Next;
end loop;
return (Left_Count = Right_Count);
end Is_Equal;
      function Extent_Of (The_Set : in Set) return Natural is
   Count : Natural := 0;
   Index : Set := The_Set;
     begin
  while Index /= null loop
    Count := Count + 1;
    Index := Index.Next;
     end loop;
return Count;
end Extent_Of;
```

```
function Is_Empty (The_Set : in Set) return Boolean is
     return (The_Set = null);
end Is_Empty;
     begin while Index /= null loop if The_Item = Index.The_Item then True;
               return True;
end if;
Index := Index.Next;
     end loop;
return False;
end Is_A_Member;
     function Is_A_Subset (Left : in Set;
   Right : in Set) return Boolean is
   Left_Index : Set := Left;
   Right_Index : Set;
     begin
          while Left_Index /= null loop
               Right_Index := Right;
while Right_Index /= null loop
if Left_Index.The_Item = Right_Index.The_Item then
                          exit;
                    exit;
else
   Right_Index := Right_Index.Next;
end if;
               end loop;
if Right_Index = null then
    return False;
               Left_Index := Left_Index.Next;
end if:
     end loop;
return True;
end Is_A_Subset;
    ease Right_Index := Right_Index.Next;
end if;
end loop;
if Right_Index = null then
                    return False;
               else
   Left_Count := Left_Count + 1;
   Left_Index := Left_Index.Next;
                end if;
          end lr;
end loop;
Right_Index := Right;
          while Right_Index /= null loop
Right_Count := Right_Count + 1;
Right_Index := Right_Index.Next;
          end loop;
     return (Left_Count < Right_Count);
end Is_A_Proper_Subset;
end Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
TYPE Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator SPECIFICATION
   GENERIC
  Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
         From_The_Set : Set,
To_The_Set : Set
     OUTFUT
To_The_Set : Set
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   END
  OPERATOR Clear
SPECIFICATION
      INPUT
      The_Set : Set
OUTPUT
The_Set : Set
      EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   END
  OPERATOR Add
SPECIFICATION
      INPUT
The_Item : Item,
To_The_Set : Set
      OUTPUT
      OUTPUT
To_The_Set : Set
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Remove
SPECIFICATION
      INPUT
         NPUT
The_Item : Item,
From_The_Set : Set
      OUTPUT
         UTPUT
From_The_Set : Set
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   END
   OPERATOR Union SPECIFICATION
      INPUT
Of_The_Set : Set,
And_The_Set : Set,
To_The_Set : Set
OUTPUT
      To_The_Set : Set
EXCEPTIONS
         Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Intersection
SPECIFICATION
INFUT
Of The Set : Set,
And The Set : Set,
To The Set : Set
OUTPUT
To The Set : Set
      EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
   OPERATOR Difference SPECIFICATION
      INPUT
          Of_The_Set : Set,
And_The_Set : Set,
```

```
To_The_Set : Set
     OUTPUT
     To_The_Set : Set EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END
  OPERATOR Is_Equal SPECIFICATION
    INPUT
Left : Set,
Right : Set
OUTPUT
       Result : Boolean
     EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Extent_Of SPECIFICATION
    INPUT
The_Set : Set
OUTPUT
       Result : Natural
     EXCEPTIONS
       Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  EMID
  OPERATOR Is_Empty
SPECIFICATION
    PECIFICATION
INPUT
The_Set : Set
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
 OPERATOR Is_A_Member
SPECIFICATION
INPUT
The_Item : Item,
Of_The_Set : Set
OUTPUT
Result
     EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR IS A Subset
SPECIFICATION
INPUT
Left : Set,
Right : Set
OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  OPERATOR Is_A_Proper_Subset SPECIFICATION
     INPUT
Left : Set,
Right : Set
     OUTPUT
     Result : Boolean
EXCEPTIONS
Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END
IMPLEMENTATION ADA
Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator
```

BINARY SEARCH

ADA SPECIFICATIONS

: out Index);

```
-- adding procedures to replace functions
     ric
type Key is limited private;
type Item is limited private;
type Index is (<>);
type Items is array(Index range <>) of Item;
with function Is_Equal (Left : in Key;
Right : in Item) return
generic
                                                                                                 procedure Location_Of (The_Key : in Key;
In_The_Items : in Items;
Result : out Index
                                                                                            -- end of modification
Boolean:
                                                                                                 with function Is_Less_Than (Left : in Key;
Right : in Item) return
                                                                                            return Index:
Boolean;
package Binary_Search is
                                                                                                 Item_Not_Found : exception;
-- modified by Tuan Nguyen
-- 20 Jan 95
                                                                                            end Binary_Search;
```

BINARY SEARCH

ADA IMPLEMENTATION

```
function Location_Of (The_Key : in Key; In_The_Items : in Items)
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
-- All Rights Reserved
-- Serial Number 0100219
                                                                                                                                           Lower_Index : Index := In_The_Items First;
Upper_Index : Index := In_The_Items Last;
The_Index : Index;
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
                                                                                                                          begin
while Lower_Index <= Upper_Index loop
The_Index :=
Index'Val((Index'Pos(Lower_Index) +
Index'Pos(Upper_Index)) / 2);
if Is_Equal(The_Key,
In_The_Items(The_Index)) then
return The_Index,
elsif Is_Less_Than(The_Key,
In_The_Items(The_Index)) then
exit when (The_Index =
In_The_Items'First);
Upper_Index := Index'Pred(The_Index)
                                                                                                                                   begin
(ii)
(11)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body Binary_Search is
     modified by Tuan Nguyen
                                                                                                                                                          Upper_Index := Index 'Pred(The_Index);
-- 20 Jan 95
-- adding procedures to replace functions
                                                                                                                           else
exit when (The_Index =
In_The_Items'Last);
       Letems'Last);
   Lower_Index := Index'Succ(The_Index);
end if;
end loop;
raise Item_Not_Found;
       begin
    Result := Location_Of(The_Key,In_The_Items);
end Location_Of;
                                                                                                                                   end Location_Of;
                                                                                                                            end Binary_Search;
-- end of modification
```

BINARY SEARCH

```
OPERATOR Location_Of
SPECIFICATION
GENERIC
           ENERIC
Key : PRIVATE_TYPE,
Item : PRIVATE_TYPE,
Index : DISCRETE_TYPE,
Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Items: ARRAY[ARRAY_EMEMBER]: Item, ARRAY_INDEX: Index],
Is_Equal: FUNCTION[Left: Key, Right: Item,
RETURN: Boolean],
Is_Less_Than: FUNCTION[Left: Key, Right: Item,
RETURN: Boolean]
```

```
The_Key : Key,
In_The_Items : Items
  OUTPUT
Result : Index
EXCEPTIONS
     Item_Not_Found
END
IMPLEMENTATION ADA Location_Of
```

BINARY INSERTION SORT

ADA SPECIFICATIONS

```
package Binary_Insertion_Sort is
    procedure Sort (The_Items : in out Items);
end Binary_Insertion_Sort;
```

BINARY INSERTION SORT

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Binary_Insertion_Sort is

procedure Sort (The_Items: in out Items) is

Temporary_Item: Item;
Left_Index: Index;
Middle_Index: Index;
Right_Index: Index;
begin
for Outer_Index in Index'Succ(The_Items'First)
.. The_Items'Last loop
Temporary_Item:= The_Items(Outer_Index);
Left_Index:= The_Items'First;
Right_Index:= Outer_Index;
while Left_Index <= Right_Index loop
```

```
Middle_Index :=
Index'Val((Index'Pos(Left Index) +
Index'Pos(Right_Index)) / 2);
if Temporary_Item <
The_Items(Middle_Index) then
    exit when (Middle_Index =
The Items'First);
                         Right_Index :=
Index Pred (Middle_Index);
else
                          exit when (Middle_Index =
Outer_Index);
Left_Index := Index 'Succ (Middle_Index);
                end if;
end loop;
if Left_Index /= Outer_Index then
The_Items(Index'Succ(Left_Index) ...
Outer Index) :=
                       The Items(Left Index ...
Index'Pred(Outer_Index));
    The_Items(Left_Index) :=
Temporary_Item;
end if;
end loop;
end Sort;
end Binary_Insertion_Sort;
```

RINARY INSERTION SORT

```
OPERATOR SOrt
SPECIFICATION
GENERIC
Item : PRIVATE_TYPE,
Index : DISCRETE_TYPE,
Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
func_*<* : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
```

```
INFUT
The_Items : Items
OUTPUT
The_Items : Items
END
IMPLEMENTATION ADA SORTE
```

BUBBLE SORT

ADA SPECIFICATIONS

```
package Bubble_Sort is
    procedure Sort (The_Items : in out Items);
end Bubble_Sort;
```

BUBBLE SORT

ADA IMPLEMENTATION

BUBBLE SORT

```
OPERATOR SOrt
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT: Item, ARRAY_INDEX:
Index],
func_*<*: FUNCTION[Left: Item, Right: Item,
RETURN: Boolean]
```

```
INPUT
The_Items: Items
OUTPUT
The_Items: Items
END
IMPLEMENTATION ADA SORT
```

HEAP SORT

ADA SPECIFICATIONS

```
package Heap_Sort is
    procedure Sort (The_Items : in out Items);
end Heap_Sort;
```

HEAP SORT

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
                   "Restricted Rights Legend"
-- Use,
    Use, duplication, or disclosure is subject to restrictions as set forth in subdivision (b) (3)
(ii)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

package body Heap_Sort is
      procedure Sort (The_Items : in out Items) is
           Temporary_Item : Item;
Left_Index : Index;
Right_Index : Index;
           The_Items(Left_Index);
The_Front
The_Back
The_Front : Index := Left_Index;
The_Back : Index :=
Index'Val(Index'Pos(The_Front) * 2);
           Index'Succ(The Back):
```

HEAP SORT

```
OPERATOR SOrt
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT: Item, ARRAY_INDEX:
Index],
func_"<": FUNCTION[Left: Item, Right: Item,
RETURN: Boolean]
```

```
INPUT
The_Items : Items
OUTPUT
The_Items : Items
END
IMPLEMENTATION ADA SORTEND
```

NATURAL MERGE SORT

ADA SPECIFICATIONS

```
with function Next_Item
return Item;
with function "<"
                                                                                                                                 (The File : in File)
generic
     eric
type Item is private;
type File is limited private;
with procedure Open_For_Reading (The_File : in out
                                                                                                                                              : in Item;
: in Item)
                                                                                                                                   Right
                                                                                     return Boolean; with function Is_End_Of_File (The_File : in File)
File);
with procedure Open_For_Writing (The_File : in out
File);
                                                                                      return Boolean:
                                                (The_File : in out
                                                                                      package Natural_Merge_Sort is
     with procedure Get
                                                                                                                (The_File : in out File;
Temporary_File_1 : in out File;
Temporary_File_2 : in out File);
                                                                                           procedure Sort (The_File
                                                The Item : out
Item):
                                                (The_File : in out
     with procedure Put
File;
                                                                                           File Is Empty : exception;
                                                The_Item : in
Item);
with procedure Close
                                                                                      end Natural_Merge_Sort;
                                                (The File : in out
File);
```

NATURAL MERGE SORT

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
    "Restricted Rights Legend"
Use, duplication, or disclosure is subject to
restrictions as set forth in subdivision (b) (3)
-- r
(11)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
package body Natural_Merge_Sort is
                           The_File : in out File;
Temporary_File_1 : in out File;
Temporary_File_2 : in out File) is
     procedure Sort (The_File
          Number_Of_Runs : Natural;
           is
                Temporary_Item : Item;
           begin
                in
Get(From_The_File, Temporary_Item);
Put(To_The_File, Temporary_Item);
if Is_End_Of_File(From_The_File) then
End_Of_Run := True;
                      End_Of_Run := (Next_Item(From_The_File)
< Temporary_Item);
end if;
           end Copy;
          procedure Copy_Run (From_The_File : in out
File:
                                     To The File : in out
File) is
                End_Of_Run : Boolean;
           begin
                loop
Copy(From_The_File, To_The_File,
End_Of_Run);
           exit when End_Of_Run;
end loop;
end Copy_Run;
          procedure Merge_Run (From_The_File : in out
File:
                                       And_The_File : in out
File:
                                       To_The_File : in out
File) is
                 End_Of_Run : Boolean;
           begin
                End_Of_Run);
                           if End_Of_Run then Copy_Run(And_The_File,
To_The_File);
                           exit;
end if;
```

```
else
                                    Copy(And_The_File, To_The_File,
End_Of_Run);
                                     if End_Of_Run then
                                           Copy_Run(From_The_File,
             end if;
end if;
end loop;
end Merge_Run;
n
To_The_File);
       begin
loop
                     open_For_Reading(The_File);
if Is_End_Of_File(The_File) then
    Close(The_File);
    Close(Temporary_File_1);
    Close(Temporary_File_2);
    raise File_Is_Empty;
                             Open_For_Writing(Temporary_File_1);
Open_For_Writing(Temporary_File_2);
                                     Copy_Run(The_File, To_The_File =>
Temporary_File_1);
                                    if not Is_End_Of_File(The_File)
                                           Copy_Run(The_File, To_The_File
=> Temporary_File_2);
                                     end if;
                             exit when Is_End_Of_File(The_File);
end loop;
Open_For_Writing(The_File);
                             Open_For_Reading(Temporary_File_1);
Open_For_Reading(Temporary_File_2);
Number_Of_Runs := 0;
Number_Or_Runs := 0;
while (not
Is_End_Of_File(Temporary_File_1)) and
(not
Is_End_Of_File(Temporary_File_2)) loop
Merge_Run(Temporary_File_1,
Temporary_File_2,
                                    To_The_File => The_File);
Number_Of_Runs := Number_Of_Runs +
1; end loop; while not

Is_End_Of_File(Temporary_File_1) loop

Copy_Run(Temporary_File_1,

To_The_File => The_File);

Number_Of_Runs := Number_Of_Runs +
                             end loop:
end loop;
while not
Is_End_Of_File(Temporary_File_2) loop
Copy_Run(Temporary_File_2,
To_The_File => The_File);
                                    Number_Of_Runs := Number_Of_Runs +
                             end loop;
              end loop;
exit when (Number_Of_Runs = 1);
end if;
end loop;
Close(The_File);
               Close (Temporary_File_1);
Close (Temporary_File_2);
        end Sort:
 end Natural_Merge_Sort;
```

NATURAL MERGE SORT

```
OPERATOR SOrt
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
File: PRIVATE_TYPE,
Open_For_Reading: PROCEDURE[The_File: in_out[t:File]],
Open_For_Writing: PROCEDURE[The_File: in_out[t:File]],
Get: PROCEDURE[The_File: in_out[t:File],
The_Item: out[t:Item]],
Put: PROCEDURE[The_File: in_out[t:File],
The_Item: in[t:Item]],
Close: PROCEDURE[The_File: in_out[t:File]],
Next_Item: FUNCTION[The_File:File, RETURN:
Item],
func_*<*: FUNCTION[Left: Item, Right: Item,
RETURN: Boolean],
```

```
Is_End_Of_File : FUNCTION[The_File : File, RETURN :
Boolean]
INFUT
  The_File : File,
  Temporary_File_1 : File,
  Temporary_File_2 : File
OUTPUT
  The_File : File,
  Temporary_File_1 : File,
  Temporary_File_2 : File
EXCEPTIONS
  File_Is_Empty
END
IMPLEMENTATION ADA Sort
END
```

ORDERED SEQUENTIAL SEARCH

ADA SPECIFICATIONS

```
generic

type Key is limited private;
type Item is limited private;
type Index is (<>);
type Index is (<>);
type Items is array(Index range <>) of Item;
with function Is_Equal (Left : in Key;
Right : in Item) return

Boolean;
with function Is_Less_Than (Left : in Key;
Right : in Item) return

Boolean;
package Ordered_Sequential_Search is

-- modified by Tuan Nguyen
-- 20 Jan 95

-- adding Index of Item;
procedulation
p
```

ORDERED SEQUENTIAL SEARCH

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
-- package body Ordered_Sequential_Search is
-- modified by Tuan Nguyen
-- 20 Jan 95
-- adding procedures to replace functions
-- procedure Location_Of (The_Key : in Key;
-- In_The_Items : in Items;
```

ORDERED SEQUENTIAL SEARCH

```
OPERATOR Location_Of
SPECIFICATION
GENERIC
Key: PRIVATE_TYPE,
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT: Item, ARRAY_INDEX:
Index],
Is_Equal: FUNCTION[Left: Key, Right: Item,
RETURN: Boolean],
Is_Less_Than: FUNCTION[Left: Key, Right: Item,
RETURN: Boolean]
```

```
INPUT
The_Key: Key,
In_The_Items: Items
OUTFUT
Result: Index
EXCEPTIONS
Item_Not_Found
END
IMPLEMENTATION ADA Location_Of
END
```

POLYPHASE SORT

ADA SPECIFICATIONS

generic		with function Next_Item	(From_The_File	e : in
Number_Of_Files : in Positive; type Item is private;		File) return Item; with function "<"	(Left	: in
type File is limited private;		Item;	21-16	: in
with procedure Open_For_Reading	(The_File : in out		Right	; in
File);	(Who File , in out	Item) return Boolean; with function Is_End_Of_File	The File	: in
with procedure Open_For_Writing	(The_File : In out	File) return Boolean;	, (*mc_r	
File); with procedure Get	(The File : in out	package Polyphase_Sort is		
File;	(Ine_File . In out	package realphase_sere is		
rile;	The_Item : out	type Files is array (1 No	mber_Of_Files)	of File;
Item);				
with procedure Put	(The_File : in out	procedure Sort (The_File		
File;	The_Item : in		iles : in out Fi : out Po	
<pre>Item);</pre>				
with procedure Close	(The_File : in out	File_Is_Empty : exception;		
File);				
		end Polyphase_Sort;		

POLYPHASE SORT

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
 -- Serial Number 0100219
-- Serial Number -- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
-- restrictions as set forth in subdivision (B) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body Polyphase_Sort is
                                          (The_File : in out File;
Temporary_Files : in out Files;
Sorted_File : out Positive)
        procedure Sort (The_File
is
Number_Of_Runs
Number_Of_Files) of Natural;
Number_Of_Files) of Natural;
Last_Item
Number_Of_Files) of Item;
                                                                       : array (1 ..
                                                                       : array (1 ...
                                                                       : array (1 ..
Number_Of_Files) of Item;
File_Map
Number_Of_Files) of Positive;
Available_Files
Number_Of_Files) of Positive;
                                                                       : array (1 ..
                                                                        : array (1 ..
                                                                        : Natural := 1:
                 Output_File : Natural := 1;
Number_Of_Available_Files : Natural;
Last_File : Positive;
                 Last Runs
                                                                          Natural:
                 Last_Dummy_Runs
                 procedure Select_File is
    Temporary_Run : Natural;
                begin

if Number_Of_Dummy_Runs(Output_File) <
    Number_Of_Dummy_Runs(Output_File + 1)
 then
                                 Output_File := Output_File + 1;
                         else
if Number_Of_Dummy_Runs(Output_File) =
0 then
                                          Level := Level + 1;
Temporary_Run := Number_Of_Runs(1);
for Index in 1 .. (Number_Of_Files
- 11 loop
- 1) loop

Number_Of_Dummy_Runs(Index) :=

Temporary_Run +

Number_Of_Runs(Index + 1) -

Number_Of_Runs(Index);

Number_Of_Runs(Index) :=

Temporary_Run +
Number_Of_Runs(Index + 1);
  end loop;
  end if;
  Output_File := 1;
                          end if:
                 end 1f;
Number_Of_Dummy_Runs(Output_File) :=
   Number_Of_Dummy_Runs(Output_File) - 1;
end Select_File;
```

```
procedure Copy_Run is
    Temporary_Item : Item;
            begin
                  loop
                        Get(The_File, Temporary_Item);
Put(Temporary_Files(Output_File),
Temporary_Item);
                         exit when (Is_End_Of_File(The_File) or
                                         (Next_Item(The_File) <
Temporary_Item));
            end loop;
Last_Item(Output_File) := Temporary_Item;
end Copy_Run;
            procedure Merge_Run is
File_Index : Fositive;
Smallest_Item : Item;
Smallest_File : Fositive;
Temporary_Item : Item;
End_Of_File : Boolean;
            begin
                  1000
                        Number_Of_Available_Files := 0;
for Index in 1 .. (Number_Of_Files - 1)
1000
                               if Number_Of_Dummy_Runs(Index) > 0
then
                                    Number_Of_Dummy_Runs(Index) :=
Number_Of_Dummy_Runs(Index)
1:
                                    Number_Of_Available_Files :=
Number_Of_Available_Files +
Available_Files(Number_Of_Available_Files) :=
    File_Map(Index);
    end if;
    end loop;
    if Number_Of_Available_Files = 0 then
Number_Of_Dummy_Runs(Number_Of_Files) :=
Smallest_File := 1;
Smallest_Item :=
Next_Item
(Temporary_Files(Available_Files(1)));
(Temporary_Files(AVAILable_Files(ITT));
    while File_Index <
Number_Of_Available_Files loop
        File_Index := File_Index +</pre>
                                           Temporary_Item :=
   Next_Item
 (Temporary_Files(Available_Files(File_Index)));
                                           if Temporary_Item
Smallest Item then
                                                 Smallest Item :=
Temporary_Item;
                                                 Smallest_File :=
File Index:
                                           end if:
                                     end loop;
```

```
Get(Temporary_Files(Available_Files(Smallest_File)),
                                   Temporary_Item);
End_Of_File :=
Is_End_Of_File
(Temporary_Files(Available_Files(Smallest_File)));
Put(Temporary_Files(File_Map(Number_Of_Files)),
                                   Temporary_Item);
if End_Of_File or else
(Next_Item
(Temporary_Files(Available_Files(Smallest_File))) < Temporary_Item) then
Available_Files(Smallest_File) :=
Available_Files(Number_Of_Available_Files);
                                            Number_Of_Available_Files
Number_Of_Available_Files - 1;
(Number_Of_Available_Files = 0);
                       end loop;
end if;
                  Last_Runs := Last_Runs - 1;
exit when (Last_Runs = 0);
end loop;
            end Merge_Run;
            Inf
for Index in 1 .. (Number_Of_Files - 1) loop
    Number_Of_Runs(Index) := 1;
    Number_Of_Dummy_Runs(Index) := 1;
    Open_For_Writing(Temporary_Files(Index));
           loop
Select_File;
Run;
                       Copy_Run;
exit when (Is_End_Of_File(The_File) or
(Output_File =
 (Number_Of_Files - 1)));
end loop;
end loop;
   while not Is_End_of_File(The_File) loop
        Select_File;
        if not (Next_Item(The_File) <
Last_Item(Output_File)) then</pre>
                             Copy_Run;
if Is_End_Of_File(The_File) then
```

```
Number_Of_Dummy_Runs(Output_File) :=
Number_Of_Dummy_Runs(Output_File) + 1;
                             else
                 else
Copy_Run;
end if;
else
Copy_Run;
end if;
end loop;
Close(The_File);
for Index in 1 . . (Number_Of_Files - 1)
1000
end loop;
loop
Last_Runs :=
Number_Of_Runs(Number_Of_Files - 1);
Number_Of_Dummy_Runs(Number_Of_Files)
Open_For_Writing(Temporary_Files(File_Map(Number_Of_Fil
                       Merge_Run;
Open_For_Reading(Temporary_Files(File_Map(Number_Of_Fil
                        Last_File := File_Map(Number_Of_Files);
Last_Dummy_Runs :=
Number_Of_Dummy_Runs(Number_Of_Files);
Last_Runs := Number_Of_Runs (Number_Of_Files - 1);
for Index in reverse 2 ...
Number Of Files loop
                             File_Map(Index) := File_Map(Index -
                             Number_Of_Runs(Index) :=
Number_Of_Runs(Index - 1) -
Last_Runs;
                       Number_Of_Dummy_Runs(Index) :=
   Number_of_Dummy_Runs(Index - 1);
end loop;
file_Map(1) := Last_File;
Number_Of_Runs(1) := Last_Runs;
Number_Of_Dummy_Runs(1) :=
Last_Dummy_Runs;
                 Runs;
   Level := Level - 1;
   exit when (Level = 0);
end loop;
for Index in 1 .. Number_Of_Files loop
Close(Temporary_Files(Index));
           end loop;
Sorted_File := File_Map(1);
end if;
      end Sort
end Polyphase Sort;
```

POLYPHASE SORT

```
OPERATOR SOrt
SPECIFICATION
GENERIC
Item : PRIVATE_TYPE,
    File : PRIVATE_TYPE,
    Open_For_Reading : PROCEDURE[The_File : in_out[t :
File]],
    Open_For_writing : PROCEDURE[The_File : in_out[t :
File]],
    Get : PROCEDURE[The_File : in_out[t : File],
    The_Item : out[t : Item]],
    Put : PROCEDURE[The_File : in_out[t : File],
    The_Item : in[t : Item]],
    Close : PROCEDURE[The_File : in_out[t : File],
    Next_Item : FUNCTION[From_The_File : File, RETURN :
Item],
```

```
func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean],
    Is_End_Of_File : FUNCTION[The_File : File, RETURN :
Boolean]
INPUT
    The_File : File,
    Temporary_Files : Files
OUTFUT
    The_File : File,
    Temporary_Files : Files,
    Sorted_File : Positive
EXCEPTIONS
    File_Is_Empty
END
IMPLEMENTATION ADA Sort
END</pre>
```

QUICK SORT

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Index is (<>);
    type Index is array(Index range <>) of Item;
    with function "<" (Left : in Item;
        Right : in Item) return Boolean;</pre>
package Quick_Sort is
    procedure Sort (The_Items : in out Items);
    end Quick_Sort;
```

OUICK SORT

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
 -- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

package body Quick_Sort is
      procedure Exchange (Left : in out Item;
Right : in out Item) is
             Temporary_Item : Item;
      cegin
   Temporary_Item := Left;
   Left := Right;
   Right := Temporary_Item;
end Exchange;
      procedure Sort (The_Items : in out Items) is
    procedure Sort_Recursive (Left_Index : in
Index;
                                                          Right_Index : in
Index) is
                    Pivot_Item : Item;
The_Front : Index;
The_Back : Index;
Middle_Index : Index;
                     if Left_Index < Right_Index then
Middle_Index :=
Index'Val((Index'Pos(Left_Index) +
 The_Items(Left_Index)):
 rne_items(left_Index);
end if;
if The_Items(Right_Index) <
The_Items(Left_Index) then
                                  Exchange (The_Items (Right_Index),
The_Items(Left_Index));
end if;
if The_Items(Right_Index) <
The_Items(Middle_Index) then
```

```
Exchange (The_Items (Right_Index),
The_Items(Middle_Index));
                       end if:
                      Pivot_Item := The_Items(Middle_Index);
Exchange(The_Items(Middle_Index),
The_Items(Index'Pred(Right_Index)));
                      The Front := Index'Succ(Left_Index);
The Back := Index'Pred(Right_Index);
if The Back /= The Items'First then
The_Back := Index'Pred(The_Back);
                      loop
while The_Items(The_Front) <
Pivot Item loop
                                The_Front :=
Index'Succ(The_Front);
                           :);
end loop;
while Pivot_Item <
Index'Pred(The Back):
                           ;
end loop;
if The_Front <= The_Back then
if (The_Front = The_Items'Last)
or else
                                      (The Back = The Items'First)
                                 return;
else
Exchange (The_Items (The_Front),
The Items(The Back));
                                       The Front :=
Index'Succ(The_Front);
                                       The_Back :=
                      end if;
end if;
exit when (The_Front > The_Back);
end loop;
Sort Benching
Index'Pred(The_Back);
           end loop;
Sort_Recursive(Left_Index, The_Back);
Sort_Recursive(The_Front, Right_Index);
end if;
end Sort_Recursive;
     begin
           Sort_Recursive(The_Items'First,
The_Items'Last);
end Sort;
end Quick_Sort;
```

QUICK SORT

```
OPERATOR Sort
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY_INDEX:
Index],
func_"<* : FUNCTION[Left: Item, Right: Item,
RETURN: Boolean]

INPUT
The_Items: Items
OUTPUT
The_Items: Items
END
IMPLEMENTATION ADA Sort
END
```

RADIX SORT

ADA SPECIFICATIONS

generic
type Item is private;
type Index is (<>);
type Items is array(Index range <>) of Item;
Number_Of_Key_Bits : in Positive;
with function Bit_Of (The_Item : in Item;

The_Bit : in Positive)
return Boolean;
package Radix_Sort is

procedure Sort (The_Items : in out Items);
end Radix_Sort;

RADIX SORT

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

package body Radix_Sort is
       procedure Sort (The_Items : in out Items) is
    procedure Sort_Recursive (Left_Index : in
                                                       Right_Index : in
Index:
                                                       Bit
                                                                       : in
Positive) is
                    Temporary_Left : Index;
Temporary_Right : Index;
Temporary_Item : Item;
Temporary_Right) loop
                                       Temporary_Left :=
Index'Succ(Temporary_Left);
     end loop;
```

```
(Bit_Of(The_Items(Temporary_Right), Bit)) and (Temporary_Left <
Temporary_Right) loop
                           Temporary_Right :=
Index'Pred(Temporary_Right);
                       end loop:
Temporary_Item :=
The_Items(Temporary_Left);
The_Items(Temporary_Left) :=
The_Items(Temporary_Right);
    The_Items(Temporary_Right) :=
Temporary_Item;
                      exit when (Temporary_Left =
Temporary_Right);
                  end loop;
if not Bit_Of(The_Items(Right_Index),
Index'Succ(Temporary_Right); end if; if Bit < Number_Of_Key_Bits then if Temporary_Right >

The_Items'First then
Bit + 1);
             end if;
         end Sort_Recursive;
    begin
         Sort_Recursive(The_Items'First, The_Items'Last,
     end Sort;
end Radix_Sort;
```

QUICK SORT

```
OPERATOR SORT
SPECIFICATION
GENERIC
Item : PRIVATE_TYPE,
Index : DISCRETE_TYPE,
Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
Bit_of : FUNCTION[The_Item : Item, The_Bit :
Positive, RETURN : Boolean]
```

```
INPUT
The_Items : Items
OUTPUT
The_Items : Items
END
IMPLEMENTATION ADA SORTEND
```

SEQUENTIAL SEARCH

ADA SPECIFICATIONS

```
generic procedure

type Key is limited private;
type Item is limited private;
type Item is limited private;
type Item is array(Index range <>) of Item;
with function Is_Equal (Left : in Key;
Right : in Item) return

Boolean;
package Sequential_Search is return Index;

-- modified by Tuan Nguyen

-- 20 Jan 95

-- adding procedures to replace functions end Sequential.
```

SEQUENTIAL SEARCH

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
-- package body Sequential_Search is
-- modified by Tuan Nguyen
-- 20 Jan 95
-- adding procedures to replace functions
-- procedure Location_Of (The_Key : in Key;
```

```
In_The_Items : in Items;
Result : out Index) is
begin
Result := Location_Of(The_Key,In_The_Items);
end Location_Of;

-- end of modification
function Location_Of (The_Key : in Key;
In_The_Items : in Items)

return Index is
begin
for The_Index in In_The_Items'Range loop
if Is_Equal(The_Key,
In_The_Items(The_Index)) then
return The_Index;
end if;
end loop;
raise Item_Not_Found;
end Location_Of;
end Sequential_Search;
```

SEQUENTIAL SEARCH

```
OPERATOR Location_Of
SPECIFICATION
GENERIC
Key: PRIVATE_TYPE,
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
Is_Equal: FUNCTION[Left : Key, Right : Item,
RETURN : Boolean]
INFUT
```

```
The_Key : Key,
In_The_Items : Items
OUTPUT
Result : Index
EXCEPTIONS
Item_Not_Found
END

IMPLEMENTATION ADA Location_Of
END
```

SHAKER SORT

ADA SPECIFICATIONS

SHAKER SORT

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
                                                                                                                            The Items (Middle Index):
                                                                                                                         The_Items(Middle_Index) :=
-- Serial Number 0100219
                                                                                             Temporary_Item;
                                                                                                             rem;
   Temporary_Index := Middle_Index;
   end if;
end loop;
ent loop;
Left_Index := Index'Succ(Temporary_Index);
for Middle_Index in Left_Index ...
                   "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
(11)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
                                                                                            package body Shaker_Sort is
     procedure Sort (The_Items : in out Items) is
   Temporary_Item : Item;
   Temporary_Index : Index;
   Left_Index : Index;
   Right_Index : Index;
}
                                                                                                                         The_Items(Middle_Index);
The_Items(Middle_Index) :=
                                                                                             Temporary_Item;
                                                                                                             tem;
    Temporary_Index := Middle_Index;
    end if;
end loop;
Right_Index := Index'Pred(Temporary_Index);
exit when (Left_Index > Right_Index);
      begin
           in
Left_Index := Index'Succ(The_Items'First);
Right_Index := The_Items'Last;
           loop
for Middle_Index in reverse Left_Index ...
end loop;
                                                                                                  end Sort;
                                                                                             end Shaker Sort:
```

SHAKER SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT: Item, ARRAY_INDEX:
Index],
func_*<*: FUNCTION[Left: Item, Right: Item,
RETURN: Boolean]

INPUT
The
OUTPUT
The
OUTP
```

INPUT The_Items : Items
OUTPUT The_Items : Items
END
IMPLEMENTATION ADA SORT
END

SHELL SORT

ADA SPECIFICATIONS

```
package Shell_Sort is
    procedure Sort (The_Items : in out Items);
end Shell_Sort;
```

SHELL SORT

ADA IMPLEMENTATION

SHELL SORT

```
OPERATOR SORT
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT: Item, ARRAY_INDEX:
Index],
func_"<": FUNCTION[Left: Item, Right: Item,
RETURN: Boolean]
```

```
The Items: Items
OUTPUT
The_Items: Items
END
IMPLEMENTATION ADA SORTEND
```

STRAIGHT INSERTION SORT

ADA SPECIFICATIONS

package Straight_Insertion_Sort is
 procedure Sort (The_Items : in out Items);
end Straight_Insertion_Sort;

STRAIGHT INSERTION SORT

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
-- package body Straight_Insertion_Sort is
-- procedure Sort (The_Items : in out Items) is
-- Temporary_Item : Item;
```

```
Inner_Index : Index;
begin
    for Outer_Index in Index'Succ(The_Items'First)
.. The_Items'Last loop
    Temporary_Item := The_Items(Outer_Index);
    Inner_Index := Outer_Index;
    while Temporary_Item <
The_Items(Index'Pred(Inner_Index)) loop
        The_Items(Inner_Index) :=
The_Items(Index'Pred(Inner_Index));
        Inner_Index := Index'Pred(Inner_Index);
    exit when (Inner_Index =
The_Items'First);
    end loop;
        red loop;
        red loop;
    end loop;
end Sort;
end Straight_Insertion_Sort;</pre>
```

STRAIGHT INSERTION SORT

PSDL

OPERATOR SORT
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT: Item, ARRAY_INDEX:
Index],
func_*<*: FUNCTION[Left: Item, Right: Item,
RETURN: Boolean]

INPUT The_Items : Items OUTPUT The_Items : Items END ITEMS : Items END ADA SOFT END

STRAIGHT SELECTION SORT

ADA SPECIFICATIONS

```
package Straight_Selection_Sort is
    procedure Sort (The_Items : in out Items);
end Straight_Selection_Sort;
```

STRAIGHT SELECTION SORT

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52,227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
-- package body Straight_Selection_Sort is

procedure Sort (The_Items : in out Items) is

Temporary_Item : Item;
Temporary_Index : Index;
```

STRAIGHT SELECTION SORT

PSDL

OPERATOR SORT
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Index: DISCRETE_TYPE,
Items: ARRAY[ARRAY_ELEMENT: Item, ARRAY_INDEX:
Index],
func_*<*: FUNCTION[Left: Item, Right: Item,
RETURN: Boolean]

INPUT The_Items : Items OUTPUT The_Items : Items END IMPLEMENTATION ADA SORTEND

STACK OJB3 SPECIFICATION

```
op underflow : -> Stack . op underflow : -> Elt .
obj STACK[X :: TRIV] is sort Stack .
   protecting NAT .
                                                                                                                                                          *** variables declaration
 *** constructors
                                        : -> Stack .
: Stack Stack -> Stack .
: Stack -> Stack .
: Elt Stack -> Stack .
: Stack -> Stack .
                                                                                                                                                                 var S S1 : Stack .
var E E1 : Elt .
       op create
      op create
op copy
op clear
op push
op pop
                                                                                                                                                          *** axioms
                                                                                                                                                               eq clear(S) = create .
eq copy(S,Sl) = S .
eq pop(create) = underflow .
eq pop(push(E,S)) = S .
eq isequal(S,Sl) = S == Sl .
eq depthof(S) = if S == create then 0 else 1 + depthof(pop(S)) fi .
eq isempty(S) = if S == create then true else false fi .
eq topof(create) = underflow .
eq topof(push(E,S)) = E .
 *** accessors
                                     : Stack Stack -> Bool .
: Stack -> Nat .
: Stack -> Bool .
: Stack -> Elt .
      op isequal
op depthof
op isempty
op topof
*** exceptions
                                                                                                                                                         endo
                                     : -> Stack .
       op overflow
```

STACK PROFILE CODES

OPERATORS	SIGNATURES	PROFILE CODES
COPY	A B -> B	3211
CLEAR	A -> A	2201
PUSH	A B -> B	3211
POP	A -> A	2201
IS_EQUAL	A B -> C	330
DEPTH_OF	A -> B	220
IS_EMPTY	A -> B	220
TOP OF	A -> B	220

SET OF PROFILE: {3211,2201,330,220}

STACK SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATION

```
Right
                                     : in Stack) return
Boolean:
    function Depth_Of (The_Stack : in Stack) return
Natural:
    function Is_Empty (The_Stack : in Stack) return
Boolean;
    function Top_Of (The_Stack : in Stack) return
Item:
    Boolean);
   procedure Iterate (Over_The_Stack : in Stack);
    Overflow : exception;
Underflow : exception;
private
    type Items is array(Positive range <>) of Item;
type Stack(The_Size : Positive) is
        record
             The_Top : Natural := 0;
The_Items : Items(1 .. The_Size);
end record;
end Stack_Sequential_Bounded_Managed_Iterator;
```

STACK SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
 Booch
-- All Rights Reserved
  -- Serial Number 0100219
 "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

 ___package body Stack_Sequential_Bounded_Managed_Iterator is
        begin
   if From_The_Stack.The_Top >
To_The_Stack.The_Size then
      raise Overflow;
else Overflow;
else
To_The_Stack.The_Items(1 ...
From_The_Stack.The_Top) :=
From_The_Stack.The_Items(1 ...
From_The_Stack.The_Top);
To_The_Stack.The_Top :=
From_The_Stack.The_Top;
end if;
end Copy;
        procedure Clear (The_Stack : in out Stack) is begin
                The_Stack.The_Top := 0;
         end Clear;
        procedure Push (The_Item : in Item;
On_The_Stack : in out Stack) is
 pegin
On_The_Stack.The_Items(On_The_Stack.The_Top +
1) := The_Item;
On_The_Stack.The_Top := On_The_Stack.The_Top +
        exception when Constraint_Error =>
                     raise Overflow;
        end Push;
        procedure Fop (The_Stack : in out Stack) is
begin
    The_Stack.The_Top := The_Stack.The_Top - 1;
         exception
when Constraint_Error =>
raise Underflow;
 -- modified by Tuan Nguyen
-- replacing procedures with functions
        procedure Is_Equal (Left : in Stack;
Right : in Stack;
Result : out Boolean) is
        Result := Is_Equal(Left,Right);
end Is_Equal;
        procedure Depth_Of (The_Stack : in Stack;
```

```
Result : out Natural) is
     begin
     Result := Depth_Of(The_Stack);
end Depth_Of;
     Result := Is_Empty(The_Stack);
end Is_Empty;
     Result := Top_Of(The_Stack);
end Top_Of;
     end of modification
     function Is_Equal (Left : in Stack; Right : in Stack) return Boolean
         if Left.The_Top /= Right.The_Top then
    return False;
else for Index in 1 .. Left.The_Top loop if Left.The_Items(Index) /=
Right.The_Items(Index) then return False;
end if;
end loop;
return True;
end if;
end Is_Equal;
     function Depth_Of (The_Stack : in Stack) return
Natural is
     begin
     return The_Stack.The_Top;
end Depth_Of;
     function Is_Empty (The_Stack : in Stack) return
Boolean
     begin
     return (The_Stack.The_Top = 0);
end Is_Empty;
     function Top_Of (The_Stack : in Stack) return Item
     begin
         return The_Stack.The_Items(The_Stack.The_Top);
     exception
when Constraint_Error =>
raise Underflow;
end Top_Of;
     procedure Iterate (Over_The_Stack : in Stack) is
    Continue : Boolean;
     begin
         for The_Iterator in reverse 1 ..
Over_The_Stack.The_Top loop
Process(Over_The_Stack.The_Items(The_Iterator),
     exit when not Continue;
end loop;
end Iterate;
 end Stack_Sequential_Bounded_Managed_Iterator;
```

STACK SEQUENTIAL BOUNDED MANAGED ITERATOR

```
TYPE Stack_Sequential_Bounded_Managed_Iterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INFUT
From_The_Stack: Stack,
To_The_Stack: Stack
OUTFUT
To_The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Clear
SPECIFICATION
INFUT
The_Stack: Stack
OUTPUT
The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Push
SPECIFICATION
INFUT
The_Item: Item,
On_The_Stack: Stack
OUTPUT
The_Itack: Stack
OUTPUT
On_The_Stack: Stack
OUTPUT
The_Item: Item,
On_The_Stack: Stack
OUTPUT
On_The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR POP
SPECIFICATION
INFUT
The_Stack: Stack
OUTPUT
The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Is_Equal
SPECIFICATION
INFUT
Left: Stack,
Right: Stack
OUTPUT
Left: Stack
OUTPUT
OUTPUT
Left: Stack
OUTPUT
OUTP
```

```
Result : Boolean
EXCEPTIONS
Overflow, Underflow
    END
 OPERATOR Depth_Of SPECIFICATION
      PECIFICATION
INPUT
The_Stack : Stack
OUTPUT
Result : Natural
EXCEPTIONS
Overflow, Underflow
    END
   OPERATOR Is_Empty
SPECIFICATION
       INPUT
       The_Stack : Stack
OUTPUT
Result : Boolean
       EXCEPTIONS
Overflow, Underflow
    END
   OPERATOR Top_Of
SPECIFICATION
INPUT
          The_Stack : Stack
       OUTPUT
Result : Item
EXCEPTIONS
          Overflow, Underflow
OPERATOR Iterate
SPECIFICATION
GENERIC
Process: PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
INPUT
Out The Stack : Stack
       Over_The_Stack : Stack
EXCEPTIONS
           Overflow, Underflow
END
IMPLEMENTATION ADA
Stack_Sequential_Bounded_Managed_Iterator
END
```

STACK SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
type Item is private;
package Stack_Sequential_Unbounded_Managed_Noniterator
is

type Stack is limited private;

procedure Copy (From_The_Stack : in Stack;
To_The_Stack : in out Stack);
procedure Clear (The_Stack : in out Stack);
procedure Push (The_Item : in Item;
On_The_Stack : in out Stack);
procedure Pop (The_Stack : in out Stack);
procedure Pop (The_Stack : in out Stack);

-- modified by Tuan Nguyen
-- replacing functions with procedures

procedure Is_Equal (Left : in Stack;
Right : in Stack;
Result : out Boolean);
procedure Depth_Of (The_Stack : in Stack;
Result : out Natural);
procedure Is_Empty (The_Stack : in Stack;
```

STACK SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
 Booch
-- All Rights Reserved
 -- Serial Number 0100219
 "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
 -- restrictions as set forth in Subdivision (B) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
 --
with Storage_Manager_Sequential;
package body
Stack_Sequential_Unbounded_Managed_Noniterator is
       type Node is
             record
                  The Item : Item:
             Next
end record;
                               : Stack;
       procedure Free (The_Node : in out Node) is
begin
             null:
       end Free;
       procedure Set_Next (The_Node : in out Node;
                                                             Stack) is
                                   To_Next : in
       begin
             The_Node.Next := To_Next;
       end Set_Next;
       function Next_Of (The_Node : in Node) return Stack
 is
       return The_Node.Next;
end Next_Of;
 package Node_Manager is new
Storage_Manager_Sequential
                                                     (Item
 Node.
                                                      Pointer
 Stack.
                               Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
       procedure Copy (From_The_Stack : in
                                                   : in Stack;
: in out Stack) is
             To_The_Stack : in out S
From_Index : Stack := From_The_Stack;
To_Index : Stack;
       exception
when Storage_Error =>
       raise Overflow;
end Copy;
      procedure Clear (The_Stack : in out Stack) is
begin
    Node_Manager.Free(The_Stack);
end Clear;
```

```
In
Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Item := The_Item;
Temporary_Node.Next := On_The_Stack;
On_The_Stack := Temporary_Node;
        exception
       when Storage_Error => raise Overflow; end Push;
       procedure Pop (The_Stack : in out Stack) is
    Temporary_Node : Stack;
begin
              In
Temporary_Node := The_Stack;
The_Stack := Temporary_Node.Next;
Temporary_Node.Next := null;
Node_Manager.Free(Temporary_Node);
       exception
when Constraint_Error =>
raise Underflow;
end Pop;
-- modified by Tuan Nguyen
-- replacing functions with procedures
       procedure Is_Equal (Left
       procedure Is_Equal (Lert : in Stack;
Right : in Stack;
Result : out Boolean);
procedure Depth_Of (The_Stack : in Stack;
Result : out Natural);
                                                               : in Stack;
       Procedure Depth_Of (The_Stack : in Stack;
Procedure Is_Empty (The_Stack : in Stack;
Result : out Nature
Result : out Noture
Procedure Top_Of (The_Stack : in Stack;
                                                              : out Boolean);
                                         Result
                                                             : out Item);
 -- end of modification
       function Is_Equal (Left : in Stack;
Right : in Stack) return Boolean
               Left_Index : Stack := Left;
Right_Index : Stack := Right;
while Left_Index /= null loop
if Left_Index.The_Item /=
Right_Index.The_Item then
                      .The_item then
    return False;
end if;
end if;
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
              end loop;
return (Right_Index = null);
        exception
               when Constraint_Error =>
       return False;
end Is_Equal;
        function Depth_Of (The_Stack : in Stack) return
Natural is
              Count : Natural := 0;
Index : Stack := The_Stack;
       begin
while Index /= rull loop
    Count := Count + 1;
    Index := Index.Next;
end loop;
    return Count;
        end Depth_Of;
        function Is_Empty (The_Stack : in Stack) return
Boolean is
       begin
        return (The_Stack = null);
end Is_Empty;
        function Top_Of (The_Stack : in Stack) return Item
       begin
               return The_Stack.The_Item;
        exception
when Constraint_Error =>
raise Underflow;
end Top_Of;
 end Stack_Sequential_Unbounded_Managed_Noniterator;
```

STACK SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

```
TYPE Stack_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
GENERIC
Litem : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
INPUT
From_The_Stack : Stack,
To_The_Stack : Stack
OUTPUT
To_The_Stack : Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Clear
SPECIFICATION
INPUT
The_Stack : Stack
OUTPUT
The_Stack : Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Push
SPECIFICATION
INPUT
The_Item : Item,
On_The_Stack : Stack
OUTPUT
On_The_Stack : Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
The_Stack : Stack
OUTPUT
The_Stack : Stack
EXCEPTIONS
OVERSTOR POP
SPECIFICATION
INPUT
The_Stack : Stack
OUTPUT
The_Stack : Stack
OUTPUT
The_Stack : Stack
OUTPUT
The_Stack : Stack
EXCEPTIONS
OVERSTOR
OVERSTO
```

STACK SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATION

```
-- end of modification

function Is_Equal (Left : in Stack;
Right : in Stack) return

Boolean;
function Depth_Of (The_Stack : in Stack) return

Natural;
function Is_Empty (The_Stack : in Stack) return

Boolean;
function Top_Of (The_Stack : in Stack) return

Item;
generic
with procedure Process (The_Item : in Item;
Continue : out

Boolean);
procedure Iterate (Over_The_Stack : in Stack);

Overflow : exception;
Underflow : exception;
private
type Node;
type Stack is access Node;
end Stack_Sequential_Unbounded_Managed_Iterator;
```

STACK SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

with Storage Manager_Sequential;
package body
Stack_Sequential_Unbounded_Managed_Iterator is
       type Node is
            record
The_Item : Item;
Wart : Stack;
             end record;
      procedure Free (The_Node : in out Node) is begin
       null;
end Free;
      procedure Set_Next (The_Node : in out Node;
To_Next : in Stack
                                                                       Stack) is
       The_Node.Next := To_Next;
end Set_Next;
       function Next_Of (The_Node : in Node) return Stack
       return The_Node.Next;
end Next_Of;
package Node_Manager is new Storage_Manager_Sequential
Node.
                                   Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
      begin
             .n
Node_Manager.Free(To_The_Stack);
if From_The_Stack /= null then
    To_The_Stack := Node_Manager.New_Item;
    To_The_Stack.The_Item :=
From_Index.The_Item:

To_Index:= To_The_Stack;
From_Index:= From_Index.Next;
while From_Index /= null loop
To_Index.Next:= Node_Manager.New_Item;
To_Index.Tem_Index.Next:= To_Index.Next;
To_Index.The_Item:

From_Index.The_Item:
From_Index.The_Item;
                           From_Index := From_Index.Next;
             end loop;
end if;
       exception
       when Storage_Error =>
raise Overflow;
end Copy;
      procedure Clear (The_Stack : in out Stack) is
begin
    Node_Manager.Free(The_Stack);
end Clear;
       Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Item := The_Item;
Temporary_Node.Next := On_The_Stack;
              On_The_Stack := Temporary_Node;
```

```
exception
              when Storage_Error => raise Overflow;
      procedure Pop (The_Stack : in out Stack) is
    Temporary_Node : Stack;
      hegin
             In
Temporary_Node := The_Stack;
The_Stack := Temporary_Node.Next;
Temporary_Node.Next := null;
Node_Manager.Free(Temporary_Node);
      exception
when Constraint_Error =>
raise Underflow;
       end Pop:
-- modified by Tuan Nguyen
-- replacing functions with procedures
      procedure Is_Equal (Left
                                         Right
                                         Result
                                                          : out Boolean) :
      procedure Depth_Of (The_Stack : in Stack;

Result : out Natural);

procedure Is_Empty (The_Stack : in Stack;
                                         Result
                                                          : out Boolean);
      procedure Top_Of (The_Stack : in Stack;
Result : out Item);
-- end of modification
      function Is_Equal (Left : in Stack;
Right : in Stack) return Boolean
              Left_Index : Stack := Left;
Right_Index : Stack := Right;
      begin
while Left_Index /= null loop
if Left_Index.The_Item /=
Right_Index.The_Item then
                    return False;
end if;
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
       end loop;
return (Right_Index = null);
exception
when Constraint_Error =>
                    return False;
       end Is_Equal;
       function Depth_Of (The_Stack : in Stack) return
Natural is

Count : Natural := 0;

Index : Stack := The_Stack;
              while Index /= null loop
Count := Count + 1;
Index := Index.Next;
       end loop;
return Count;
end Depth_Of;
       function Is_Empty (The_Stack : in Stack) return
Boolean is
begin
       return (The_Stack = null);
end Is_Empty;
       function Top_Of (The_Stack : in Stack) return Item
              return The_Stack.The_Item;
      return ins________
exception
when Constraint_Error =>
raise Underflow;
end Top_Of;
       procedure Iterate (Over_The_Stack : in Stack) is
    The_Iterator : Stack := Over_The_Stack;
    Continue : Boolean;
       begin
  while not (The_Iterator = null) loop
    Process(The_Iterator.The_Item, Continue);
                     exit when not Continue;
The_Iterator := The_Iterator.Next;
       end loop;
end Iterate;
```

end Stack_Sequential_Unbounded_Managed_Iterator;

STACK SEQUENTIAL UNBOUNDED MANAGED ITERATOR

```
TYPE Stack_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
INPUT
From_The_Stack: Stack
OUTPUT
To_The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Clear
SPECIFICATION
INPUT
The_Stack: Stack
OUTPUT
The_Stack: Stack
OUTPUT
The_Stack: Stack
OUTPUT
The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Push
SPECIFICATION
INPUT
The_Item: Item,
On_The_Stack: Stack
OUTPUT
On_The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
The_Item: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
The_Stack: Stack
OUTPUT
The_Stack: Stack
EXCEPTIONS
Overflow, Underflow
END

OPERATOR IS_Equal
SPECIFICATION
INPUT
Left: Stack,
Right: Stack
CUTPUT
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATION

```
generic
type Item is private;
package
Stack_Sequential_Unbounded_Unmanaged_Noniterator is

type Stack is limited private;

procedure Copy (From_The_Stack : in Stack;
To_The_Stack : in out Stack);
procedure Clear (The_Stack : in out Stack);
procedure Push (The_Item : in Item;
On_The_Stack : in out Stack);
procedure Pop (The_Stack : in out Stack);

-- modified by Tuan Nguyen
-- replacing functions with procedures

procedure Is_Equal (Left : in Stack;
Right : in Stack;
Result : out Boolean);
procedure Depth_Of (The_Stack : in Stack;
Result : out Natural);
procedure Is_Empty (The_Stack : in Stack;
```

```
Result : out Boolean);
procedure Top_Of (The_Stack : in Stack;
Result : out Item);

-- end of modification

function Is_Equal (Left : in Stack;
Right : in Stack) return

Boolean;
function Depth_Of (The_Stack : in Stack) return

Natural;
function Is_Empty (The_Stack : in Stack) return

Boolean;
function Top_Of (The_Stack : in Stack) return

Item;
Overflow : exception;
Underflow : exception;

private
type Node;
type Stack is access Node;
end Stack_Sequential_Unbounded_Unmanaged_Noniterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
                  "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52,227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

package body
Stack_Sequential_Unbounded_Unmanaged_Noniterator is
     type Node is
           record
                 The_Item : Item;
Next : Stack;
           end record:
     if From_The_Stack = null then
    To_The_Stack := null;
To_The_Stack := new Node'(The_Item => From_Index.The_Item,
mill):
                 To_Index := To_The_Stack;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
          Nex

To_Index := To_Index.Next;
From_Index := From_Index.Next;
end loop;
end if;
sption
                                                            Next
null);
     exception
           when Storage_Error => raise Overflow;
     end Copy;
     procedure Clear (The_Stack : in out Stack) is begin
           The_Stack := null;
     end Clear:
     On_The_Stack := new Node'(The_Item => The_Item,
Next =>
On_The_Stack);
     exception
when Storage_Error =>
raise Overflow;
     end Push:
     procedure Pop (The_Stack : in out Stack) is
```

```
raise Underflow;
       end Pop;
 -- modified by Tuan Nguyen
-- replacing functions with procedures
       procedure Is_Equal (Left
                                                       : in Stack:
       procedure Is_Equal (Lert : in Stack; Right : out Boolean); procedure Depth_Of (The_Stack : in Stack; Result : out Natural);
      Result : out Natural)
procedure Is_Empty (The_Stack : in Stack;
Result : out Boolean);
procedure Top_Of (The_Stack : in Stack;
Result : out Item);
 -- end of modification
       function Is_Equal (Left : in Stack;
    Right : in Stack) return Boolean
             Left_Index : Stack := Left;
Right_Index : Stack := Right;
begin
    while Left_Index /= null loop
    if Left_Index.The_Item /=
Right_Index.The_Item then
        return False;
    end if;
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
    end loop:
       begin
       end loop;
return (Right_Index = null);
exception
when Constraint_Error =>
                   return False:
       end Is_Equal;
       function Depth_Of (The_Stack : in Stack) return
 Natural is
             Count : Natural := 0;
Index : Stack := The_Stack;
       begin
             while Index /= null loop
             Count := Count + 1;
Index := Index.Next;
end loop;
return Count;
       end Depth_Of;
       function Is_Empty (The_Stack : in Stack) return
 Boolean is
       return (The_Stack = null);
end Is_Empty;
       function Top_Of (The_Stack : in Stack) return Item
             return The_Stack.The_Item;
       exception when Constraint_Error =>
                    raise Underflow;
       end Top_Of;
 end Stack_Sequential_Unbounded_Unmanaged_Noniterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
TYPE Stack_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
GENERIC
    Item : FRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
INPUT
    From_The_Stack : Stack,
    To_The_Stack : Stack
OUTFUT
    To_The_Stack : Stack
EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Clear
SPECIFICATION
INFUT
    The_Stack : Stack
OUTPUT
    The_Stack : Stack
OUTPUT
    The_Stack : Stack
EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Push
SPECIFICATION
INPUT
    The_Item : Item,
    On_The_Stack : Stack
OUTPUT
    On_The_Stack : Stack
EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
    The_Stack : Stack
EXCEPTIONS
    Overflow, Underflow
END

OPERATOR Pop
SPECIFICATION
INPUT
    The_Stack : Stack
OUTPUT
    The_Stack : Stack
OUTPUT
    The_Stack : Stack
OUTPUT
    The_Stack : Stack
EXCEPTIONS
    Overflow, Underflow
END
```

```
OPERATOR Is_Equal
SPECIFICATION
INFUT
Left: Stack,
Right: Stack
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Depth_Of
SPECIFICATION
INFUT
The_Stack: Stack
OUTPUT
Result: Natural
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Stack: Stack
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Stack: Stack
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Underflow
END

OPERATOR Top_Of
SPECIFICATION
INFUT
The_Stack: Stack
OUTPUT
Result: Item
EXCEPTIONS
Overflow, Underflow
END

END

END

END

END

END

IMPLEMENTATION ADA
Stack_Sequential_Unbounded_Unmanaged_Noniterator
END
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATION

```
generic
type Item is private;
package Stack_Sequential_Unbounded_Unmanaged_Iterator
is

type Stack is limited private;

procedure Copy (From_The_Stack : in Stack;
To_The_Stack : in out Stack);
procedure Clear (The_Stack : in out Stack);
procedure Push (The_Item : in Item;
On_The_Stack : in out Stack);
procedure Pop (The_Stack : in out Stack);

-- modified by Tuan Nguyen
-- replacing functions with procedures

procedure Is_Equal (Left : in Stack;
Right : in Stack;
Result : out Boolean);
procedure Depth_of (The_Stack : in Stack;
Result : out Natural);
procedure Top_Of (The_Stack : in Stack;
Result : out Boolean);
procedure Top_Of (The_Stack : in Stack;
Result : out Stack;
```

```
-- end of modification

function Is_Equal (Left : in Stack;
Right : in Stack) return

Boolean;
function Depth_Of (The_Stack : in Stack) return

Natural;
function Is_Empty (The_Stack : in Stack) return

Boolean;
function Top_Of (The_Stack : in Stack) return

Item;
generic
with procedure Process (The_Item : in Item;
Continue : out

Boolean);
procedure Iterate (Over_The_Stack : in Stack);

Overflow : exception;
Underflow : exception;

private
type Node;
type Stack is access Node;
end Stack_Sequential_Unbounded_Unmanaged_Iterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)

    (ii)
    of the rights in Technical Data and Computer
    Software Clause of FAR 52.227-7013. Manufacturer:
    Wizard software, 2171 S. Parfet Court, Lakewood,
    Colorado 80227 (1-303-987-1874)

package body
Stack_Sequential_Unbounded_Unmanaged_Iterator is
       type Node is
            record
The_Item : Item;
Text : Stack;
             end record;
      begin
   if From_The_Stack = null then
        To_The_Stack := null;
else
To_The_Stack := new Node'(The_Item =>
From_Index.The_Item,
                                                              Next
                    To_Index := To_The_Stack;
From_Index := From_Index.Next;
while From_Index /= null loop
To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
            Nex
    To_Index := To_Index.Next;
    From_Index := From_Index.Next;
end loop;
end if;
ppio-
null);
      end 11,
exception
when Storage_Error =>
raise Overflow;
       end Copy;
      procedure Clear (The_Stack : in out Stack) is begin
             The_Stack := null;
      procedure Push (The_Item : in Item;
On_The_Stack : in out Stack) is
             On_The_Stack := new Node'(The_Item => The_Item,
Next =>
On_The_Stack);
      he_Stack,,
exception
when Storage_Error =>
raise Overflow;
       procedure Pop (The_Stack : in out Stack) is begin
             The_Stack := The_Stack.Next;
       exception
when Constraint_Error =>
raise Underflow;
       end Pop;
```

```
modified by Tuan Nguyen replacing functions with procedures
       procedure Is_Equal (Left
                                                         · in Stack:
      Right : in Stack;
Result : out Boolear
procedure Depth_Of (The_Stack : in Stack;
                                                         : in Stack;
: out Boolean);
      procedure Depth_Of (The_Stack : In Stack;
Result : out Natural)
procedure Is_Empty (The_Stack : in Stack;
Result : out Boolean);
procedure Top_Of (The_Stack : in Stack;
Result : out Item);
                                                           : out Natural):
-- end of modification
       function Is_Equal (Left : in Stack;
Right : in Stack) return Boolean
             Left_Index : Stack := Left;
Right_Index : Stack := Right;
       begin
while Left_Index /= null loop
if Left_Index.The_Item /=
Right_Index.The_Item then
                    .Tne_item then
return False;
end if;
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
             end loop;
return (Right_Index = null);
       exception when Constraint_Error =>
       return False;
end Is_Equal;
       function Depth_Of (The_Stack : in Stack) return
Natural is

Count : Natural := 0;

Index : Stack := The_Stack;
              while Index /= null loop
Count := Count + 1;
Index := Index.Next;
       end loop;
return Count;
end Depth_Of;
       function Is_Empty (The_Stack : in Stack) return
Boolean is
begin
       return (The_Stack = null);
end Is_Empty;
      function Top_Of (The_Stack : in Stack) return Item
              return The_Stack.The_Item;
       exception
when Constraint_Error =>
raise Underflow;
       end Top_Of;
      procedure Iterate (Over_The_Stack : in Stack) is
    The_Iterator : Stack := Over_The_Stack;
    Continue : Boolean;
             while not (The_Iterator = null) loop
    Process(The_Iterator.The_Item, Continue);
       exit when not Continue;
The_Iterator := The_Iterator.Next;
end loop;
end Iterate;
 end Stack_Sequential_Unbounded_Unmanaged_Iterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
OUTPUT
Result : Boolean
       EXCEPTIONS
Overflow, Underflow
   END
   OPERATOR Depth_Of SPECIFICATION
      PECIFICATION
INPUT
The_Stack : Stack
OUTPUT
Result : Natural
EXCEPTIONS
         Overflow, Underflow
   END
   OPERATOR Is_Empty
SPECIFICATION
INPUT
The_Stack : Stack
       OUTPUT
Result : Boolean
EXCEPTIONS
   Overflow, Underflow
   OPERATOR Top_Of
SPECIFICATION
      INPUT
The_Stack : Stack
OUTFUT
Result : Item
EXCEPTIONS
          Overflow, Underflow
   OPERATOR Iterate
   OPERATOR Iterate
SPECIFICATION
GENERIC
Process: PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
INPUT
Over_The_Stack : Stack
EXCEPTIONS
         Overflow, Underflow
   END
IMPLEMENTATION ADA
Stack_Sequential_Unbounded_Unmanaged_Iterator
```

STORAGE MANAGER SEQUENTIAL

ADA SPECIFICATION

generic
type Item is limited private;
type Pointer is access Item;
with procedure Free (The_Item : in out
Item);
with procedure Set_Pointer (The_Item : in out
Item;
The_Pointer : in
Pointer);
with function Pointer_Of (The_Item : in Item)
return Pointer;
package Storage_Manager_Sequential is

procedure Free (The_Pointer : in out Pointer);
-- modified by Tuan Nguyen
-- replace function with procedure
procedure New_Item (Result : Pointer);
-- end of modification

function New_Item return Pointer;
end Storage_Manager_Sequential;

STORAGE MANAGER SEQUENTIAL

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
-- Serial Number 0100219
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
-- package body Storage_Manager_Sequential is

Free_List : Pointer := null;

procedure Free (The_Pointer : in out Pointer) is
    Temporary_Pointer : Pointer;
    begin
    while The_Pointer /= null loop
        Temporary_Pointer := The_Pointer;
        The_Pointer := Pointer_Of(The_Pointer.all);
        Free_Ctemporary_Pointer.all);

The_Pointer := Free_List);
    Free_List := Temporary_Pointer;
```

```
end loop;
end Free;

-- modified by Tuan Nguyen
-- replace function with procedure
   procedure New_Item (Result : Pointer) is
   begin
        Result := New_Item;
end New_Item;

-- end of modification

function New_Item return Pointer is
        Temporary_Pointer : Pointer;
begin
        if Free_List = null then
            return new Item;
        else
            Temporary_Pointer := Free_List;
            Free_List :=
Pointer_Of(Temporary_Pointer.all);
            Set_Pointer(Temporary_Pointer.all,
The_Pointer => null);
            return Temporary_Pointer;
        end if;
end New_Item;
end Storage_Manager_Sequential;
```

STORAGE MANAGER SEQUENTIAL

PSDL

OPERATOR Free
SPECIFICATION
GENERIC
Item: PRIVATE_TYPE,
Pointer: ACCESS_TYPE,
Free: PROCEDURE[The_Item: in_out[t: Item]],
Set_Pointer: PROCEDURE[The_Item: in_out[t: Item],
The_Pointer: in[t: Pointer]],
Pointer_Of: FUNCTION[The_Item: Item, RETURN: Pointer]

INPUT
The Pointer : Pointer
OUTPUT
The Pointer : Pointer
END
IMPLEMENTATION ADA Free
END

STRING SEQUENTIAL UNBOUNDED CONTROLLED ITERATOR

ADA SPECIFICATIONS

```
in String;
out Natural);
in String;
                                                                                                                                                             (The_String
                                                                                                                       procedure Length_Of
                                                                                                                                                              Result
      type Item is private;
type Item is private;
type Substring is array(Positive range <>) of Item;
with function "<" (Left : in Item;
with function "<" (Left : in Item) return Boolean;
package String_Sequential_Unbounded_Controlled_Iterator is</pre>
                                                                                                                       procedure Is_Null
                                                                                                                                                             (The String
                                                                                                                                                                                          out Boolean);
in String;
in Positive;
                                                                                                                                                             Result
(The_String
                                                                                                                       procedure Item Of
                                                                                                                                                             At_The_Position
Result
(The_String
                                                                                                                                                                                          out Item);
in String;
out Substring);
                                                                                                                       procedure Substring_Of
      type String is limited private;
                                                                                                                                                              Result
                                                                                                                                                             (The_String
From_The_Position
To_The_Position
                                                                                                                       procedure Substring_Of
                                                                                                                                                                                          in String:
     procedure Copy
                                 (From The String
                                                                          String;
                                                                                                                                                                                          in Positive;
in Positive;
out Substring);
                                 To_The_String
(From_The_Substring
To_The_String
                                                                in out String);
     procedure Copy
                                                                in Substring;
in out String);
     procedure Clear
procedure Prepend
                                                                in out String);
                                 (The String
                                                                                                                  -- end of modification
                                 (The_String
To_The_String
                                                                in
                                                                          String:
                                                                in out String);
in Substring;
                                                                                                                                                                                      : in String:
                                                                                                                       function Is Equal
                                                                                                                                                            (Left
     procedure Prepend
                                 (The Substring
                                                                                                                                                             Right
                                                                                                                                                                                      : in String)
                                                                                                                                                                                                             return
                                 To_The_String
(The_String
To_The_String
                                                                in out String);
                                                                in String;
in out String);
                                                                                                                  Boolean:
     procedure Append
                                                                                                                                                            Left
                                                                                                                                                                                      : in Substring:
                                                                                                                        function Is_Equal
                                                                                                                                                                                      : in String)
                                                                                                                                                                                                             return
     procedure Append
                                 (The_Substring
To_The_String
                                                                in
                                                                          Substring:
                                                                    out String);
                                                                                                                  Boolean;
                                                                                                                        function Is_Equal
                                                                                                                                                            (Left
                                                                                                                                                                                      : in String:
     procedure Insert
                                 (The String
                                                                in
                                                                          String:
                                                                                                                                                             Right
                                                                                                                                                                                      : in Substring) return
                                  In_The_String
At_The_Position
                                                                in out String:
                                                                                                                  Boolean:
                                                                                                                                                            (Left
                                                                                                                        function Is_Less_Than
                                                                                                                                                                                      · in String:
     procedure Insert
                                 (The Substring
                                                                in
                                                                          Substring:
                                                                                                                                                             Right
                                                                                                                                                                                                             return
                                                                    out String;
Positive);
                                  In_The_String
                                  At The Position
                                                                in
                                                                                                                                                            (Left
                                                                                                                                                                                      : in Substring;
                                                                                                                       function Is_Less_Than
     procedure Delete
                                 (In_The_String
From_The_Position
                                                                in out String;
                                                                                                                                                             Right
                                                                                                                                                                                                             return
                                                                          Positive:
                                                                          Positive);
                                                                                                                  Boolean;
                                  To The Position
                                                                in
                                                                                                                                                            (Left
                                                                                                                                                                                      : in String;
: in Substring) return
                                                                                                                       function Is_Less_Than
                                 (In_The_String
At_The_Position
With_The_String
     procedure Replace
                                                                in out String;
                                                                                                                                                             Right
                                                                in
in
                                                                          Positive;
String);
                                                                                                                                                           (Lef+
                                                                                                                                                                                      : in String;
: in String)
                                                                                                                       function Is_Greater_Than
     procedure Replace (In_The_String At_The_Position With_The_Substring procedure Set_Item (In_The_String
                                                                in out
                                                                          String:
                                                                                                                                                                                                             return
                                                                          Positive;
Substring);
                                                                in
                                                                in
in out
                                                                                                                  Boolean:
                                                                                                                                                                                      : in Substring;
                                                                          String;
Positive;
                                                                                                                                                           (Left
                                                                                                                       function Is_Greater_Than
                                                                                                                                                                                      : in String)
                                                                                                                                                                                                             return
                                  At_The_Position
With_The_Item
                                                                in
                                                                          Item):
                                                                                                                  Boolean;
                                                                                                                                                                                      : in String;
: in Substring)
                                                                                                                       function Is Greater_Than
                                                                                                                                                                                                            return
                                                                                                                                                             Right
                                                                                                                  Boolean;
function Length_Of
     modified by Vincent Hong and Tuan Nguyen
date: 9 April 1995
adding procedures to replace functions
                                                                                                                                                            (The_String
                                                                                                                                                                                      : in String)
                                                                                                                                                                                                             return
                                                                                                                  Natural:
                                                                                                                                                           (The_String
                                                                                                                                                                                      : in String)
                                                                                                                       function Is_Null
                                                                                                                  Boolean;
function Item_Of
                                                                        in String;
     procedure Is_Equal
                                           (Left
                                                                                                                                                           (The_String
At_The_Position
                                                                                                                                                                                      : in String;
: in Positive) return
                                           Right
Result
(Left
                                                                        in String:
                                                                        out Boolean);
in Substring;
in String;
     procedure Is Equal
                                                                                                                                                                                      : in String) return
                                                                                                                       function Substring_Of
                                                                                                                                                           (The String
                                            Right
                                                                        out Boolean);
in String;
in Substring;
                                                                                                                 Substring;
function Substring_Of
                                           Result
(Left
                                                                                                                                                                                        in String;
in Positive;
                                                                                                                                                           (The_String
                                                                                                                                                            The_String
From_The_Position : in Positive;
in Positive)return
     procedure Is Equal
                                            Right
                                           Result
(Left
                                                                        out Boolean);
                                                                        in String;
in String;
                                                                                                                  Substring:
     procedure Is_Less_Than
                                            Right
                                                                        out Boolean);
in Substring;
in String;
out Boolean);
in String;
                                            Result
                                                                                                                       (Left
     procedure Is_Less_Than
                                            Right
                                            Result
     procedure Is Less Than
                                                                        in Substring;
out Boolean);
in String;
                                                                                                                       Overflow : exception;
Position_Error : exception;
                                            Right
                                            Result
     procedure Is_Greater_Than (Left
                                                                        in String;
out Boolean);
in Substring;
in String;
                                                                                                                       type Structure is access Substring;
type String is
record
     procedure Is_Greater_Than
                                           (Left
     Right
Result
procedure Is_Greater_Than (Left
                                                                                                                                  The_Length : Natural := 0;
The_Items : Structure;
                                                                      : out Boolean);
: in String;
: in Substring;
: out Boolean);
                                            Right
                                                                                                                  end String_Sequential_Unbounded_Controlled_Iterator;
```

STRING SEQUENTIAL UNBOUNDED CONTROLLED ITERATOR

ADA IMPLEMENTATION

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                    "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
with Storage_Manager_Sequential;
package body String_Sequential_Unbounded_Controlled_Iterator is
      type Node Pointer is access Node; type Node is
            record
                  The_Structure : Structure;
Next : Node_Pointer;
            end record;
      type Header_Pointer is access Header;
      type Header is
            record
                  The_Size : Natural;
The_Structures : Node_Pointer;
Next : Header_Pointer;
            end record:
      procedure Free(The_Node : in out Node) is
begin
            The_Node.The_Structure := null;
     procedure Set_Next (The_Node : in out Node;
To_Next : in Node_
                                                                  Node_Pointer) is
            The_Node.Next := To_Next;
      end Set_Next;
      function Next_Of (The_Node : in Node) return Node_Pointer is
      begin
            return The_Node.Next;
      package Node Manager is new Storage_Manager_Sequential
                                                          age_manager_sequential
(Item => Node,
Pointer => Node_Pointer,
Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
                                                          (Item
     procedure Free(The_Header : in out Header) is begin
            The_Header.The_Size := 0;
      end Free;
     procedure Set_Next (The_Header : in out Header;
To_Next : in Header_
                                                                     Header_Pointer) is
      begin
            The Header.Next := To_Next;
      end Set_Next;
      function Next_Of (The_Header : in Header) return Header_Pointer is
      return The_Header.Next;
end Next_Of;
      package Header_Manager is new Storage_Manager_Sequential
                                                             (Item => Header,
Pointer => Header_Po
Free => Free,
Set_Pointer => Set_Next,
                                                                                => Header,
=> Header_Pointer,
=> Free,
                                                              Pointer_Of
                                                                                => Next_Of);
      task Structure Manager is
            entry Free (The_Structure : in out Structure);
entry Get_New_Structure (The_Size : in Natural;
The_Structure : out Structure);
      end Structure_Manager;
      task body Structure_Manager is
Free_List : Header_Pointer;
The_Structure : Structure;
            Node_Index : Node_Pointer;
Previous_Header : Header_Pointer;
Header_Index : Header_Pointer;
           loop
begin
select
ac
      begin
                               accept Free (The_Structure : in out Structure) do
                                     Previous_Header := null;
Header_Index := Free_List;
```

```
while Header_Index /= null loop
   if The_Structure'Length <
      Header_Index.The_Size then
      exit;
   elsif The_Structure'Length =</pre>
                                                 Header_Index.The_Size then
Node_Index := Node_Manager.New_Item;
Node_Index.The_Structure :=
The Structure:
                                                     Node_Index.Next :=
Header Index. The Structures:
                                                     Header Index. The Structures :=
Node Index:
                                                     The_Structure := null;
                                               return;
end if;
                                              Previous_Header := Header_Index;
Header_Index := Header_Index.Next;
                                       Header_Index := Header_Index.Next;
end loop;
Header_Index := Header_Manager.New_Item;
Header_Index := Node_Manager.New_Item;
Node_Index := Node_Manager.New_Item;
Node_Index.The_Structure := The_Structure;
Header_Index.The_Structures := Node_Index;
if Previous_Header = null then
Header_Index.Next := Free_List;
                                              Free_List := Header_Index;
                                       else
Header_Index.Next := Previous_Header.Next;
Previous_Header.Next := Header_Index;
                                end ir;
The_Structure := null;
end Free;
                          or
                                 accept Get_New_Structure (The_Size
                                                                                                   : in
                                                                            The_Structure : out
Structure) do
                                       Previous_Header := null;
Header_Index := Free_List;
while Header_Index /= null loop
   if Header_Index.The_Size >= The_Size then
    Node_Index :=
Header_Index.The_Structures;
                                                     Header_Index.The_Structures :=
Node Index.Next:
                                                    Node_Index.Next := null;
if Header_Index.The_Structures = null
then
                                                           if Previous_Header = null then
    Free_List :=
Header_Index.Next:
                                                                  Previous_Header.Next :=
Header_Index.Next;
                                                           end if;
Header_Index.Next := null;
                                                            Header_Manager.Free(Header_Index);
                                                     The_Structure :=
Node_Index.The_Structure;
                                                     Node_Manager.Free(Node_Index);
                                               return;
end if;
                                              Previous_Header := Header_Index;
Header_Index := Header_Index.Next;
                                        end loop;
                                        The_Structure := new Substring(1 .. The_Size);
                                 end Get_New_Structure;
                                 terminate:
                           end select;
                    exception
when Storage_Error =>
                                 nu11:
      end;
end loop;
end Structure_Manager;
      procedure Free (The_Structure : in out Structure) is
begin
   if The_Structure /= null then
             Structure_Manager.Free(The_Structure); end if;
       function New_Structure (The_Size : in Natural) return Structure is
             Temporary_Structure : Structure;
      begin
Structure_Manager.Get_New_Structure(The_Size,
Temporary_Structure);
return Temporary_Structure;
end New_Structure;
      procedure Set (The_String : in
    To_The_Size : in
    Preserve_The_Value : in
    Temporary_Structure : Structure;
                                                              : in out String:
                                                                              Natural:
                                                                              Boolean) is
```

```
if To_The_Size = 0 then
    Free(The_String.The_Items);
elsif The_String.The_Items = null then
    The_String.The_Items := New_Structure(The_Size =>
To The Size):
             elsif To_The_Size > The_String.The_Items'Length then
                   if To_The_Size > The_String.The_Items Length then
    Temporary_Structure := New_Structure(To_The_Size);
    Temporary_Structure(1 .. The_String.The_Length) :=
        The_String.The_Items(1 .. The_String.The_Length);
    Free(The_String.The_Items);
    The_String.The_Items := Temporary_Structure;

                          end if:
             end if;
The_String.The_Length := To_The_Size;
       exception
when Storage_Error =>
      raise Overflow;
end Set;
      procedure Copy (From_The_String : in String; To_The_String : in out String) is
      begin
             Set(To_The_String,
    To_The_Size => From_The_String.The_Length,
    Preserve_The_Value => False);
To_The_String.The_Items(1 .. From_The_String.The_Length) :=
    From_The_String.The_Items(1 .. From_The_String.The_Length);
       exception
              when Storage_Error => raise Overflow;
       end Conv:
      procedure Copy (From_The_Substring : in Substring; To_The_String : in out String) is
                Set(To_The_String,
       exception
              when Storage_Error => raise Overflow;
       end Copy:
       procedure Clear (The_String : in out String) is
       begin
             Set (The String,
                    To_The_Size => 0,
Preserve_The_Value => False);
       exception
             when Storage_Error =>
raise Overflow;
       end Clear:
      procedure Prepend (The_String : in String;
To_The_String : in out String) is
Old_Length : Natural := To_The_String.The_Length;
New_Length : Natural :=
To_The_String.The_Length + The_String.The_Length;
             ns

Set(To_The_String, => New_Length,

To_The_Size => New_Length,

Preserve_The_Value => True);

To_The_String.The_Items((The_String.The_Length + 1) ...
New Length)
             ;: To_The_String.The_Items(1 .. Old_Length);
To_The_String.The_Items(1 .. The_String.The_Length) :=
The_String.The_Items(1 .. The_String.The_Length);
       exception
              when Storage_Error =>
       end Prepend;
      procedure Prepend (The_Substring : in Substring;
To_The_String : in out String) is
Old_Length : Natural := To_The_String.The_Length;
New_Length : Natural :=
To_The_String.The_Length + The_Subst
                                  To_The_String.The_Length + The_Substring'Length;
             New_Length)
             := To_The_String.The_Items(1 .. Old_Length);
To_The_String.The_Items(1 .. The_Substring'Length) :=
 The Substring;
        exception
when Storage_Error =>
raise Overflow;
       end Prepend;
       begin
```

```
:= The_String.The_Items(1 .. The_String.The_Length);
       exception
when Storage_Error =>
                     raise Overflow;
      To_The_String.The_Length + The_Substring'Length;
       begin
              Set (To The String,
             To_The_Size => New_Length,
Preserve_The_Value => True);
To_The_String.The_Items((Old_Length + 1) .. New_Length)
                   = The_Substring;
       exception
when Storage_Error =>
                     raise Overflow;
       end Append;
                                     (The_String : in String;
In_The_String : in out String;
At_The_Position : in Positive) is
: Natural := In_The_String.The_Length;
       procedure Insert (The_String
              Old Length
              New_Length
                                      : Natural :=
                                         In_The_String.The_Length +
The_String.The_Length;
End_Position : Natural :=
                                        At_The_Position + The_String.The_Length;
              if At_The_Position > In_The_String.The_Length then
  raise Position_Error;
                     Set(In_The_String,
	To_The_Size => New_Length,
	Preserve_The_Value => True);
In_The_String.The_Items(End_Position .. New_Length) :=
	In_The_String.The_Items(At_The_Position .. Old_Length);
In_The_String.The_Items(At_The_Position .. (End_Position -
                         The_String.The_Items(1 .. The_String.The_Length);
              end if:
       exception
when Storage_Error =>
raise Overflow;
       end Insert:
       procedure Insert (The_Substring : in Substring;
In_The_String : in out String;
At_The_Position : in Positive) is
                                     At_The_Position : in Positive) is
: Natural := In_The_String.The_Length;
: Natural :=
In_The_String.The_Length +
              New Length
The_Substring'Length;
End_Position : Natural :=
At_The_Position + The_Substring'Length;
              in
if At_The_Position > In_The_String.The_Length then
    raise Position_Error;
              else
                     Set(In_The_String,
                     Set(In_The_String,
To_The_Size => New_Length,
Preserve_The_Value => True);
In_The_String.The_Items(End_Position .. New_Length) :=
In_The_String.The_Items(At_The_Position .. Old_Length);
In_The_String.The_Items(At_The_Position .. (End_Position
1)) :=
                         The Substring:
              and if:
       end 1.,
exception
when Storage_Error =>
raise Overflow;
       procedure Delete (In_The_String : in out String;
From_The_Position : in Positive;
To_The_Position : in Positive) is
New_Length : Natural;
       begin
              In if (From_The_Position > In_The_String.The_Length) or else (To_The_Position > In_The_String.The_Length) or else (From_The_Position > To_The_Position) then
                      raise Position_Error;
                     New_Length := In_The_String.The_Length -
(To_The_Position - From_The_Position + 1);
In_The_String.The_Items(From_The_Position .. New_Length)
                     In_The_String.The_Items
  ((To_The_Position + 1) ... In_The_String.The_Length);
Set(In_The_String,
   To_The_Size => New_Length,
   Preserve_The_Value => True);
               end if:
        exception
when Storage_Error =>
raise Overflow;
       procedure Replace (In_The_String : in out String;
At_The_Position : in Positive
With_The_String : in String)
```

begin

```
if (At_The_Position > In_The_String.The_Length) or else
    (End_Position > In_The_String.The_Length) then
    raise Position_Error;
                                                                                                                         result := Length_Of
end Length_Of;
                                                                                                                                                                    (The_String);
                                                                                                                                                               (The_String Result
                                                                                                                                                                                          : in String;
: out Boolean) is
                 In_The_String.The_Items(At_The_Position .. End_Position)
                                                                                                                         procedure Is Null
                    With_The_String.The_Items(1 ..
With_The_String.The_Length);
    end if;
end Replace;
                                                                                                                         result := Is_Null
end Is_Null;
                                                                                                                                                                    (The_String);
                                                                                                                                                               (The_String
At_The_Position
Result
                                                                                                                         procedure Item Of
                                                                                                                                                                                          · in String:
                                                                                                                                                                                          : in Positive;
: out Item) is
     procedure Replace (In_The_String : in out String;
At_The_Position : in Positiv
With_The_Substring : in Substri
                                                                         Substring) is
                                                                                                                         begin
                                                                                                                        result := Item_Of
end Item_Of;
                                                                                                                                                                    (The_String, At_The_Position);
           (The_String Result
                                                                                                                                                                                         : in String;
: out Substring) is
                                                                                                                         procedure Substring Of
     begin
           if (At_The_Position > In_The_String.The_Length) or else
(End_Position > In_The_String.The_Length) then
raise Position_Error;
                                                                                                                         result := Substring_Of
end Substring_Of;
                                                                                                                                                                    (The String):
                 In_The_String.The_Items(At_The_Position .. End_Position)
                                                                                                                         procedure Substring Of
                                                                                                                                                                                         : in String:
                                                                                                                                                               (The String
                                                                                                                                                                The_Position : in Fositive;
To_The_Position : in Positive;
Result : out Substring) is
                    With The Substring:
     end if;
end Replace;
                                                                                                                         begin
                                                                                                                   pegin
    result :=
Substring_Of(The_String,From_The_Position,To_The_Position);
end Substring_Of;
     procedure Set_Item (In_The_String : in out String;
At_The_Position : in Positiv
With_The_Item : in Item) i
                                                                      Positive;
Item) is
           if At_The_Position > In_The_String.The_Length then
    raise Position_Error;
                                                                                                                        end of modification
           else
In_The_String.The_Items(At_The_Position) := With_The_Item;
                                                                                                                        function Is_Equal (Left : in String;
Right : in String) return Boolean is
                                                                                                                        begin
if Left.The_Length /= Right.The_Length then
      end if;
end Set_Item;
                                                                                                                                    return False;
     modified by Vincent Hong and Tuan Nguyen
date: 9 April 1995
adding procedures to replace functions
                                                                                                                       .1..Lei
.srt.The_Items
return False;
end if;
end loop;
return True;
end if;
end Is_Equal;
function J
                                                                                                                              else
for Index in 1 .. Left.The_Length loop
if Left.The_Items(Index) /= Right.The_Items(Index)
                                                                                                                   then
      procedure Is_Equal
                                                                      : in String;
: in String;
                                             Right
                                                                       : out Boolean) is
                                             Result
                                                 (Left, Right);
           result := Is_Equal
      end Is_Equal;
                                                                                                                         function Is_Equal (Left : in Substring;
Right : in String) return Boolean is
      procedure Is_Equal
                                                                       : in Substring;
                                             Right
                                                                       : in String;
: out Boolean) is
                                             Result
                                                                                                                              in
if Left'Length /= Right.The_Length then
    return False;
     result := Is_Equal
end Is_Equal;
                                                 (Left.Right):
                                                                                                                              else
for Index in 1 .. Left'Length loop
if Left(Left'First + Index - 1) /=
                                                                      : in String;
: in Substring;
: out Boolean) is
                                            (Left
      procedure Is_Equal
                                                                                                                  Right
Result
      begin
      result := Is_Equal end Is_Equal;
                                                 (Left.Right);
      procedure Is_Less_Than
                                            (Left
                                                                       : in String:
                                             Right
Result
                                                                       · in String:
                                                                                                                        function Is_Equal (Left : in String; Right : in Substring) return Boolean is
                                                                       : out Boolean) is
      begin
     result := Is_Less_Than (Left,Right);
end Is_Less_Than;
                                                                                                                         begin
                                                                                                                              if Left.The_Length /= Right'Length then
    return False;
                                                                                                                              else
for Index in 1 .. Left.The_Length loop
if Left.The_Items(Index) /= Right(Right'First + Index
                                                                       : in Substring;
      procedure Is_Less_Than
                                            (Left
                                                                                                                       return False;
end if;
end loop;
return True;
end if;
end s;
end if;
end if;
                                             Right
Result
                                                                       : in String:
                                                                       : out Boolean) is
                                                                                                                   - 1) then
      begin
      result := Is_Less_Than end Is_Less_Than;
                                                 (Left, Right);
                                                                       : in String;
: in Substring;
                                            (Left
      procedure Is Less Than
                                             Right
Result
                                                                       out Boolean) is
      begin
                                                                                                                         function Is_Less_Than (Left : in String; Right : in String) return Boolean is
      result := Is_Less_Than end Is_Less_Than;
                                                (Left, Right);
                                                                                                                         begin
                                                                                                                              in
for Index in 1 .. Left.The_Length loop
   if Index > Right.The_Length then
        return False;
elsif Left.The_Items(Index) < Right.The_Items(Index) then</pre>
                                                                       : in String;
: in String;
      procedure Is_Greater_Than (Left
                                             Right
                                                                       : out Boolean) is
                                             Result
      result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
                                                                                                                                    return True;
elsif Right.The_Items(Index) < Left.The_Items(Index) then
return False;
end if;
      procedure Is_Greater_Than (Left
                                                                       · in Substring:
                                                                                                                         end loop;
return (Left.The_Length < Right.The_Length);
end Is_Less_Than;
                                             Right
Result
                                                                       : in String;
: out Boolean) is
      result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
                                                                                                                         function Is_Less_Than (Left : in Substring;
Right : in String) return Boolean is
                                                                                                                         begin
      procedure Is_Greater_Than (Left
                                                                       : in String:
                                                                                                                   for Index in 1 .. Left'Length loop
if Index > Right.The_Length then
return False;
elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
                                             Right
Result
                                                                       : in Substring;
: out Boolean) is
      result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
                                                                                                                                    return True;
elsif Right.The_Items(Index) < Left(Left'First + Index -
                                                                       : in String;
      procedure Length Of
                                            (The String
```

: out Natural) is

Result

```
return False;
end if;
       end loop;
end loop;
return (Left'Length < Right.The_Length);
end Is_Less_Than;</pre>
       function Is_Less_Than (Left : in String;
Right : in Substring) return Boolean is
       begin
               in
for Index in 1 .. Left.The_Length loop
if Index > Right'Length then
    return False;
elsif Left.The_Items(Index) < Right(Right'First + Index -</pre>
                      return True;
elsif Right(Right'First + Index - 1) <</pre>
Left.The_Items(Index) then
                      return False;
end if;
       end loop;
return (Left.The_Length < Right'Length);
end Is_Less_Than;
       function Is_Greater_Than (Left : in String;
Right : in String) return Boolean is
               for Index in 1 .. Left.The_Length loop
if Index > Right.The_Length then
                      return True;
elsif Left.The_Items(Index) < Right.The_Items(Index) then
    return False;
elsif Right.The_Items(Index) < Left.The_Items(Index) then
    return True;
end if;</pre>
       end loop;
return False;
end Is_Greater_Than;
       function Is_Greater_Than (Left : in Substring;
Right : in String) return Boolean is
       begin
begin
for Index in 1 .. Left'Length loop
if Index > Right.The_Length then
return True;
elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
return False;
elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
       return True;
end if;
end loop;
return False;
end Is_Greater_Than;
       function Is_Greater_Than (Left : in String; Right : in Substring) return Boolean is
       begin
             in
  for Index in 1 .. Left.The_Length loop
    if Index > Right'Length then
        return True;
    elsif Left.The_Items(Index) < Right(Right'First + Index -</pre>
1) then
                             return False:
```

```
elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
    return True;
    end if;
    end loop;
    return False;
    end Is_Greater_Than;</pre>
      function Length_Of (The_String : in String) return Natural is
      begin
           return The String. The Length;
      end Length_Of;
      function Is_Null (The_String : in String) return Boolean is
      return (The_String.The_Length = 0);
end Is_Null;
      if At_The_Position > The_String.The_Length then
    raise Position_Error;
else
           return The_String.The_Items(At_The_Position); end if;
      end Item_Of;
      function Substring_Of (The_String : in String) return Substring is
    Temporary_Structure : Substring(1 .. 1);
      begin
           return The_String.The_Items(1 .. The_String.The_Length);
      exception
           when Constraint_Error =>
     return Temporary_Structure(1 .. 0); end Substring_Of;
      function Substring_Of (The_String : in String;
From_The_Position : in Positive;
To_The_Position : in Positive) return
Substring is
    begin

if (From_The_Position > The_String.The_Length) or else

(To_The_Position > The_String.The_Length) or else

(From_The_Position > To_The_Position) then

raise Position_Error;
           else
                return The_String.The_Items(From_The_Position ...
To_The_Position);
end if;
end Substring_Of;
     procedure Iterate (Over_The_String : in String) is
   Continue : Boolean;
     begin
           for The_Iterator in 1 .. Over_The_String.The_Length loop
Process(Over_The_String.The_Items(The_Iterator),
Continue):
                exit when not Continue:
           end loop;
      end Iterate;
end String_Sequential_Unbounded_Controlled_Iterator;
```

STRING SEQUENTIAL UNBOUNDED CONTROLLED ITERATOR

TYPE String_Sequential_Unbounded_Controlled_Iterator SPECIFICATION
<pre>GENERIC Item : PRIVATE_TYPE, Substring : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive], func_*c* : FUNCTION[Left : Item, Right : Item, RETURN : Boolean] OPERATOR Copy</pre>
SPECIFICATION INPUT From_The_String : String,
To_The_String : String OUTPUT To_The_String : String
EXCEPTIONS Overflow, Position_Error END
OPERATOR COPY SPECIFICATION INPUT
From The Substring : Substring, To The String : String OUTPUT
To_The_String : String EXCEPTIONS Overflow, Position_Error END
OPERATOR Clear SPECIFICATION
INPUT The_String : String OUTPUT
The_String : String EXCEPTIONS Overflow, Position_Error END
OPERATOR Prepend SPECIFICATION
INPUT The_String : String, To_The_String : String
OUTPUT To_The_String : String EXCEPTIONS Overflow, Position_Error
END
OPERATOR Prepend SPECIFICATION INPUT The_Substring : Substring,
To_The_String : String OUTPUT To_The_String : String EXCEPTIONS
Overflow, Position_Error
OPERATOR Append SPECIFICATION INPUT
The_String : String, To_The_String : String OUTPUT To_The_String : String
EXCEPTIONS Overflow, Position_Error END
OPERATOR Append SPECIFICATION INPUT
The_Substring : Substring, To_The_String : String OUTFUT
To_The_String : String EXCEPTIONS Overflow, Position_Error END
OPERATOR Insert
SPECIFICATION INFUT The_String: String, In_The_String: String,
At_The_Position : Positive OUTFUT In_The_String : String
EXCEPTIONS Overflow, Position_Error END
OPERATOR Insert SPECIFICATION INPUT
The Substring : Substring, In_The_String : String,

```
At_The_Position : Positive
OUTPUT In_The_String : String
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Delete
SPECIFICATION
INPUT
In_The_String : String,
From_The_Position : Positive,
To_The_Position : Positive
OUTPUT
 To_The_Position : Position
OUTPUT
In_The_String : String
EXCEPTIONS
Overflow, Position_Error
END
 OPERATOR Replace
SPECIFICATION
INPUT
       INPUT
In_The_String : String,
At_The_Fosition : Positive,
With_The_String : String
OUTPUT
In_The_String : String
EXCEPTIONS
Overflow, Position_Error
  END
 OPERATOR Replace
SPECIFICATION
INPUT
In_The_String: String,
At_The_Position: Positive,
With_The_Substring: Substring
OUTPUT
In_The_String: String
       In_The_String : String
EXCEPTIONS
Overflow, Position_Error
  END
 OPERATOR Set_Item
SPECIFICATION
INPUT
In_The_String: String,
At_The_Position: Positive,
With_The_Item: Item
OUTPUT
In_The_String: String
EXCEPTIONS
OVERFLOW_Position_Error
             Overflow, Position_Error
 OPERATOR IS_Equal SPECIFICATION
       INPUT
Left : String,
Right : String
       Result : Boolean
EXCEPTIONS
Overflow, Position_Error
  END
  OPERATOR IS_Equal SPECIFICATION
       INPUT
Left : Substring,
Right : String
        OUTPUT
       Result : Boolean
EXCEPTIONS
Overflow, Position_Error
  END
  OPERATOR Is_Equal
SPECIFICATION
INPUT
Left: String,
Right: Substring
  Right : Substring
OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
END
  OPERATOR Is_Less_Than
SPECIFICATION
INFUT
Left : String,
Right : String
OUTPUT
RESULT : Boolean
FYCEPTIONS
        EXCEPTIONS
              Overflow, Position_Error
```

```
OPERATOR Is_Less_Than
OPERATOR Is_Less_Than
SPECIFICATION
INPUT
Left: Substring,
Right: String
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Position_Error
 OPERATOR Is_Less_Than SPECIFICATION
          INPUT
Left : String,
Right : Substring
          Right . Substitute South State State
 END
  OPERATOR Is Greater Than
 SPECIFICATION
INPUT
Left: String,
            Right : String
OUTPUT
Result : Boolean
            EXCEPTIONS
                      Overflow, Position_Error
  END
 OPERATOR Is_Greater_Than SPECIFICATION
            INPUT
Left : Substring,
           Right : String
OUTPUT
Result : Boolean
            EXCEPTIONS
                        Overflow, Position_Error
OPERATOR Is_Greater_Than
SPECIFICATION
INPUT
Left : String,
           Right : Substring
OUTPUT
Result : Boolean
EXCEPTIONS
                        Overflow, Position_Error
OPERATOR Length_Of
 SPECIFICATION
INPUT
The_String : String
            OUTPUT
                        Result : Natural
```

```
EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR IS_Null SPECIFICATION
      INPUT
  INPUT
The_String : String
OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
END
   OPERATOR Item_Of SPECIFICATION
     PECIFICATION
INFUT
The String : String,
At_The_Position : Positive
      OUTPUT
Result : Item
EXCEPTIONS
        Overflow, Position_Error
   OPERATOR Substring_Of
   SPECIFICATION
INPUT
The_String : String
      OUTPUT
        Result : Substring
      EXCEPTIONS
         Overflow, Position_Error
   OPERATOR Substring_Of SPECIFICATION
     The String: String,
From The Position: Positive,
To The Position: Positive
OUTPUT
Result: Substring
EXCEPTIONS
         Overflow, Position_Error
   OPERATOR Iterate
SPECIFICATION
GENERIC
        Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
      INPUT
Over_The_String : String
EXCEPTIONS
        Overflow, Position_Error
   END
END IMPLEMENTATION ADA String_Sequential_Unbounded_Controlled_Iterator END
```

STRING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
procedure Length Of
                                                                                                                                                           (The String
                                                                                                                                                                                      : in String:
generic
     ric
type Item is private;
type Substring is array(Positive range <>) of Item;
with function "<" (Left : in Item;
Right : in Item) return Boolean;
Right : in Item return Boolean;
Right : Rebounded Managed_Iterator is
                                                                                                                                                           Result
(The_String
Result
(The_String
                                                                                                                                                                                        out Natural);
in String;
out Boolean);
                                                                                                                      procedure Is Null
                                                                                                                      procedure Item Of
                                                                                                                                                                                        in String:
                                                                                                                                                           At_The_Position
Result
(The_String
                                                                                                                                                                                        in Positive:
package String_Sequential_Unbounded_Managed_Iterator is
                                                                                                                                                                                        out Item);
in String;
     type String is limited private;
                                                                                                                      procedure Substring Of
                                                                                                                                                            Result
                                                                                                                                                                                        out Substring) :
                                                                                                                                                           (The_String
From_The_Position
                                                                                                                                                                                        in String;
in Positive;
                                                                                                                      procedure Substring Of
     procedure Copy
                                (From_The_String
                                                                         String:
                                To_The_String
(From_The_Substring
                                                               in out String);
in Substring;
                                                                                                                                                            To_The_Position
Result
                                                                                                                                                                                        in Positive
     procedure Copy
                                                                                                                                                                                        out Substring);
                                                               in out String);
                                  To The String
                                (The_String
(The_String
To_The_String
                                                               in out String);
in String;
in out String);
     procedure Clear
                                                                                                                -- end of modification
     procedure Prepend
                                To_The_String
(The_Substring
To_The_String
(The_String
To_The_String
To_The_String
                                                               in Substring; in out String; in String;
                                                                                                                      function Is Equal
                                                                                                                                                                                    : in String;
     procedure Prepend
                                                                         Substring:
                                                                                                                                                                                                          return
                                                                                                                                                           Right.
                                                                                                                                                                                    : in String)
                                                                                                                Boolean;
function Is_Equal
     procedure Append
                                                                                                                                                                                      in Substring;
                                                                                                                                                          (Left
                                                               in out String);
                                                                                                                                                                                      in String)
                                                               in
in
in
                                                                                                                                                          Right
                                                                         Substring:
     procedure Append
                                                                                                                Boolean:
                                                                                                                                                                                      in String;
in Substring) return
                                                                                                                      function Is_Equal
                                                                                                                                                         (Left
     procedure Insert
                                (The String
                                                                         String;
                                                                                                                                                          Right
                                In_The_String
At_The_Position
(The_Substring
                                                               in out String;
in Positive
in Substri
                                                                                                                Boolean:
                                                                          Substring;
                                                                                                                      function Is_Less_Than
                                                                                                                                                         (Left
                                                                                                                                                                                      in String:
     procedure Insert
                                                                                                                                                                                                          return
                                In_The_String
At_The_Position
(In_The_String
From_The_Position
                                                               in out String:
                                                                    Positive);
out String;
Positive;
                                                               in
in
in
                                                                                                                Boolean:
                                                                                                                                                         (Left
                                                                                                                                                                                      in Substring;
                                                                                                                      function Is Less Than
     procedure Delete
                                                                                                                                                          Right
                                                                                                                                                                                    : in String)
                                                                                                                                                                                                          return
                                To_The_Position
(In_The_String
At_The_Position
With_The_String
                                                               in
in
in
in
                                                                         Positive):
                                                                                                                Boolean;
                                                                                                                                                                                    : in String;
: in Substring) return
                                                                                                                                                         (Left
                                                                   out String;
Positive;
                                                                                                                      function Is Less Than
     procedure Replace
                                                                                                                                                          Right
                                                                                                                Boolean;
function Is_Greater_Than (Left
Right
                                                                         String);
                                (In_The_String
At_The_Position
With_The_Substring
     procedure Replace
                                                               in out String:
                                                               in Positive
in Substring;
in out String;
                                                                         Positive:
                                                                                                                Boolean:
                                                                                                                                                                                    : in Substring;
                                                                                                                                                         (Left
Right
                                                                                                                      function Is_Greater_Than
     procedure Set_Item (In_The_String
                                                                                                                                                                                                          return
                                  At_The_Position
With_The_Item
                                                               in
                                                                         Positive:
                                                                         Item):
                                                                                                                Boolean:
                                                                                                                      function Is_Greater_Than
                                                                                                                                                         (Left
                                                                                                                                                                                    : in String:
                                                                                                                                                                                    : in Substring) return
     modified by Vincent Hong and Tuan Nguyen
date: 9 April 1995
adding procedures to replace functions
                                                                                                                Boolean:
                                                                                                                                                         (The_String
                                                                                                                                                                                    · in String)
                                                                                                                      function Length Of
                                                                                                                Natural;
function Is_Null
                                                                                                                                                                                    : in String)
                                                                                                                                                         (The_String
                                                                                                                                                                                                          return
                                           (Left
                                                                                                                Boolean;
     procedure Is_Equal
                                                                       in String:
                                                                                                                                                                                    : in String;
: in Positive) return
                                                                                                                                                         (The_String
                                                                        in String;
out Boolean);
in Substring;
                                                                                                                      function Item Of
                                                                                                                                                           At_The_Position
                                            Result
     procedure Is_Equal
                                           (Left
                                                                                                                Item;
                                           Right
Result
                                                                                                                                                                                    : in String) return
                                                                                                                      function Substring_Of
                                                                       in String;
out Boolean);
in String;
in Substring;
                                                                                                                Substring;
function Substring_Of
                                                                                                                                                         (The_String
                                                                                                                                                                                    : in String:
     procedure Is Equal
                                                                                                                                                          From_The_Position : in Positive;
To_The_Position : in Positive)return
                                           Right
Result
                                                                        out Boolean):
                                                                        in String;
in String;
out Boolean);
                                                                                                                Substring:
     procedure Is_Less_Than
                                            Right
                                                                                                                      Result
     procedure Is_Less_Than
                                           (Left
                                                                        in Substring:
                                                                        in String;
out Boolean);
in String;
                                            Result
     procedure Is Less Than
                                           (Left
                                                                                                                      Overflow : exception;
Position_Error : exception;
                                          Right
Result
(Left
                                                                        in Substring;
                                                                        out Boolean):
                                                                        in String;
in String;
     procedure Is Greater Than
                                            Right
                                                                                                                      type Structure is access Substring;
                                            Result
                                                                        out Boolean);
                                                                        in Substring;
                                           (Left
                                                                                                                      type String is
     procedure Is Greater_Than
                                                                        in String;
out Boolean);
                                                                                                                           record
                                            Right
                                                                                                                                 The_Length : Natural := 0;
The_Items : Structure;
                                            Result
                                          (Left
     procedure Is_Greater_Than
                                                                        in String;
in Substring;
                                                                                                                            end record:
                                                                                                                 end String_Sequential_Unbounded_Managed_Iterator;
                                            Result
```

STRING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE String_Sequential_Unbounded_Managed_Iterator SPECIFICATION
   GENERIC
      Item : PRIVATE_TYPE,
      ! FKIVATE_TYPE,
Substring : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive],
func_'<* : FUNCTION[Left : Item, Right : Item, RETURN : Boolean]</pre>
   OPERATOR CODY
      FECTION INPUT
From_The_String : String,
To_The_String : String
OUTPUT
To_The_String : String
      EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR Copy
SPECIFICATION
      INPUT
From_The_Substring : Substring,
To_The_String : String
      OUTPUT
To_The_String : String
      EXCEPTIONS
          Overflow, Position_Error
   END
   OPERATOR Clear
SPECIFICATION
      INPUT
      The_String : String
OUTPUT
The_String : String
      EXCEPTIONS
          Overflow, Position_Error
   OPERATOR Prepend
      PECIFICATION
INPUT
         NPUT
The_String : String,
TO_The_String : String
      OUTPUT
To_The_String : String
      EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR Prepend
SPECIFICATION
INPUT
         NPUT
The_Substring : Substring,
To_The_String : String
      OUTPUT
To_The_String : String
      EXCEPTIONS
          Overflow, Position_Error
  OPERATOR Append
SPECIFICATION
INPUT
The String: String,
To_The_String: String
OUTPUT
TO_The_String: String
EXCEPTIONS
Control of the String Profession From
          Overflow, Position_Error
   END
   OPERATOR Append
SPECIFICATION
INPUT
         The_Substring : Substring,
To_The_String : String
      OUTPUT
To_The_String : String
EXCEPTIONS
          Overflow, Position_Error
   בתאים
   OPERATOR Insert
   SPECIFICATION
      PECIFICATION
INPUT
The_String: String,
In_The_String: String,
At_The_Position: Positive
OUTPUT
In_The_String: String
      EXCEPTIONS
          Overflow, Position_Error
   OPERATOR Insert
   SPECIFICATION
INPUT
```

WFUT
The_Substring : Substring,
In_The_String : String,

```
At_The_Position : Positive
   OUTPUT
In_The_String : String
   EXCEPTIONS
       Overflow, Position_Error
OPERATOR Delete
SPECIFICATION
   INPUT
      In_The_String : String,
From_The_Position : Positive,
To_The_Position : Positive
   OUTPUT
   In_The_String : String
EXCEPTIONS
       Overflow, Position_Error
OPERATOR Replace SPECIFICATION
   INPUT
      NFUI
In_The_String : String,
At_The_Position : Positive,
With_The_String : String
   OUTPUT
In_The_String : String
EXCEPTIONS
       Overflow, Position_Error
OPERATOR Replace
OPERATOR Replace
SPECIFICATION
INPUT
In_The_String : String,
At_The_Position : Positive,
With_The_Substring : Substring
  With_____OUTPUT
In_The_String : String
EXCEPTIONS
Overflow, Position_Error
OPERATOR Set_Item
OPERATOR Set_Item
SPECIFICATION
INFUT
In_The_String : String,
At_The_Position : Positive,
With_The_Item : Item
   OUTPUT
      In The String : String
   EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Equal SPECIFICATION
   INPUT
      Left : String,
Right : String
   OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
OPERATOR Is_Equal SPECIFICATION
   INPUT
   Left : Substring,
Right : String
OUTPUT
Result : Boolean
   EXCEPTIONS
Overflow, Position_Error
END
OPERATOR IS_Equal SPECIFICATION
   INPUT
      Left : String,
Right : Substring
   OUTPUT
Result : Boolean
       Overflow, Position_Error
OPERATOR Is_Less_Than SPECIFICATION
   INPUT
      NPUT
Left : String,
Right : String
      Result : Boolean
   EXCEPTIONS
       Overflow, Position_Error
```

```
OPERATOR Is_Less_Than
SPECIFICATION
INFUT
Left : Substring,
Right : String
OUTFUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Less_Than SPECIFICATION
   INPUT
Left : String,
Right : Substring
    OUTPUT
    OUTFUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
 END
OPERATOR Is_Greater_Than SPECIFICATION
    INPUT
Left : String,
Right : String
Right: String
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Greater_Than SPECIFICATION
    INPUT
    Left : Substring,
Right : String
OUTPUT
    OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
 END
OPERATOR Is_Greater_Than SPECIFICATION
    INPUT
Left : String,
Right : Substring
    OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Position_Error
 END
OPERATOR Length_Of SPECIFICATION
    INPUT
The_String : String
OUTPUT
        Result : Natural
```

```
EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR IS_Null SPECIFICATION
      INPUT
The_String : String
OUTPUT
  OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
END
   OPERATOR Item_Of SPECIFICATION
      PECIFICATION
INPUT
The_String : String,
At_The_Position : Positive
OUTPUT
Result : Item
EXCEPTIONS
         Overflow, Position_Error
   EMD
   OPERATOR Substring_Of
SPECIFICATION
INPUT
The_String : String
OUTPUT
  OUTPUT
Result: Substring
EXCEPTIONS
Overflow, Position_Error
END
   OPERATOR Substring_Of SPECIFICATION
      PECIFICATION
INPUT
The_String : String,
From_The_Position : Positive,
To_The_Position : Positive
      Result : Substring
EXCEPTIONS
Overflow, Position_Error
   END
   OPERATOR Iterate
   SPECIFICATION
GENERIC
Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean!
      INPUT
Over_The_String : String
EXCEPTIONS
         Overflow, Position_Error
   END
END
IMPLEMENTATION ADA String_Sequential_Unbounded_Managed_Iterator END
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

return

return

return

return

return

return

return

```
: in Substring:
                                                                                                                                                    Right
Result
generic
                                                                                                                                                                               out Boolean);
     type Item is private;
type Substring is array(Positive range <>) of Item;
with function "<" (Left : in Item;
                                                                                                                procedure Length_Of
                                                                                                                                                    (The String
                                                                                                                                                                               in String;
                                                                                                                                                    Result
(The_String
                                                                                                                                                                                out Natural);
                              (Left : in Item;
Right : in Item) return Boolean;
                                                                                                                                                                               in String;
out Boolean);
                                                                                                                procedure Is_Null
                                                                                                                                                     Result
package String_Sequential_Unbounded_Unmanaged_Noniterator is
                                                                                                                                                    (The_String
At_The_Position
                                                                                                                                                                               in String:
                                                                                                                procedure Item_Of
                                                                                                                                                                               in Positive;
out Item);
     type String is limited private;
                                                                                                                                                     Result
                                                         : in String;
: in out String);
: in Substring;
: in out String);
: in out String);
: in String;
                                                                                                                                                    (The String
                                                                                                                                                                               in String:
                               (From_The_String
To_The_String
(From_The_Substring
                                                                                                                procedure Substring_Of
     procedure Copy
                                                                                                                                                                                out Substring):
                                                                                                                                                    Result
(The_String
                                                                                                                                                                               in String;
in Positive;
                                                                                                                procedure Substring_Of
     procedure Copy
                                                                                                                                                    From_The_Position
To_The_Position
Result
                               To_The_String
(The_String
                                                                                                                                                                               in Positive
     procedure Clear
     procedure Prepend
                               (The String
                               To_The_String
(The_Substring
To_The_String
                                                            in out String);
                                                                                                               end of modification
                                                            in Substring in out String);
                                                                      Substring;
     procedure Prepend
                                                                                                                                                                           : in String;
                                                                                                                                                  (Left
                                                                                                                 function Is Equal
                               (The_String
To_The_String
(The_Substring
     procedure Append
                                                            in
                                                                     String;
                                                                                                                                                                           : in String)
                                                            in out String);
in Substring;
                                                                                                           Boolean:
     procedure Append
                                                                                                                                                                              in Substring;
                                                                                                                 function Is_Equal
                                                                                                                                                  (Left
                               To_The_String
(The_String
In_The_String
                                                            in out String);
                                                            in String,
in out String;
Positive);
                                                                                                                                                                           : in String)
     procedure Insert
                                                                                                           Boolean:
                                                                                                                                                  (Left
                                                                                                                                                                              in String:
                                                                                                                 function Is_Equal
                               At_The_Position
(The_Substring
                                                                                                                                                                           : in Substring) return
                                                                      Substring;
     procedure Insert
                                                            in out String;
in Positive);
                                In The String
                               In_The_String
At_The_Position
(In_The_String
From_The_Position
To_The_Position
(In_The_String
                                                                                                                 function Is_Less_Than
                                                                                                                                                  (Left
                                                                                                                                                                           : in String:
                                                                                                                                                                           : in String)
                                                                                                                                                   Right
                                                                out String;
Positive;
     procedure Delete
                                                                                                           Boolean:
                                                            in
                                                                                                                                                                              in Substring;
                                                                                                                 function Is_Less_Than
                                                                                                                                                  (Left
                                                            in
                                                                      Positive):
                                                                                                                                                                            : in String)
                                                            in
in
                                                                     String;
Positive;
                                                                out
     procedure Replace
                                                                                                           Boolean:
                                At The Position
                                                                                                                                                                           : in String:
                                                                                                                                                  (Left
                                                                                                                 function Is_Less_Than
                               With_The_String
(In_The_String
At_The_Position
                                                            in
                                                                      String);
                                                                                                                                                                           : in Substring) return
                                                            in
in
in
                                                                out String;
Positive;
Substring);
     procedure Replace
                                                                                                           Boolean;
                                                                                                                                                  (Left
                               With_The_Substring
(In_The_String
At_The_Position
With_The_Item
                                                                                                                function Is Greater_Than
                                                                                                                                                    Right
                                                                                                                                                                           : in String)
     procedure Set_Item
                                                            in out String;
                                                                     Positive;
Item);
                                                                                                           Boolean;
                                                                                                                                                                             in Substring;
in String)
                                                                                                                function Is_Greater_Than
                                                                                                                                                   Right
                                                                                                                                                                           : in String:
                                                                                                                function Is_Greater_Than
     modified by Vincent Hong and Tuan Nguyen
                                                                                                                                                                              in Substring) return
     date: 9 April 1995
adding procedures to replace functions
                                                                                                                                                                           · in String)
                                                                                                                function Length_Of
                                                                                                                                                  (The_String
                                                                                                           Natural:
     procedure Is Equal
                                                                                                                                                                           : in String)
                                                                                                                                                  (The_String
                                                                                                                 function Is_Null
                                                                    in String;
                                         Right
                                                                                                           Boolean;
function Item_Of
                                                                                                                                                                           : in String;
: in Positive
                                        Result
(Left
                                                                    out Boolean);
                                                                                                                                                  (The_String
At_The_Position
                                                                    in Substring;
in String;
out Boolean);
     procedure Is Equal
                                                                                                                                                                              in Positive) return
                                         Right
                                                                                                           Item;
   function Substring_Of
                                         Result
                                                                                                                                                                           : in String) return
                                                                                                                                                  (The String
                                         (Left
Right
     procedure Is_Equal
                                                                    in String;
                                                                    in Substring;
out Boolean);
                                                                                                                                                  (The_String : From The_Position :
                                                                                                                                                                             in String;
in Positive;
                                                                                                                function Substring Of
                                         Result
     procedure Is_Less_Than
                                         (Left
                                                                    in String:
                                                                                                                                                                           : in Positive) return
                                                                                                                                                   To_The_Position
                                         Right
Result
                                                                    in String;
out Boolean);
in Substring;
                                                                                                           Substring:
     procedure Is_Less_Than
                                         (Left
                                                                                                                                       exception:
                                        Right
Result
(Left
                                                                    in String;
                                                                                                                 Position_Error : exception;
                                                                    out Boolean);
     procedure Is_Less_Than
                                        Right
Result
(Left
                                                                    in Substring;
                                                                                                                 type Structure is access Substring;
                                                                    out Boolean):
                                                                    in String;
in String;
                                                                                                                 type String is
     procedure Is_Greater_Than
                                                                                                                      record
                                         Right
                                                                                                                           The_Length : Natural := 0;
The_Items : Structure;
                                          Result
                                                                    out Boolean);
     procedure Is_Greater_Than
                                                                    in Substring;
                                                                    in String;
                                                                                                                      end record;
                                          Right
                                                                                                           end String_Sequential_Unbounded_Unmanaged_Noniterator;
                                                                    out Boolean);
                                          Result
     procedure Is_Greater_Than (Left
                                                                  : in String:
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                   *Restricted Rights Legend*
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body String_Sequential_Unbounded_Unmanaged_Noniterator is
     procedure Set (The_String : in
    To_The_Size : in
    Preserve_The_Value : in
    Temporary_Structure : Structure;
                                                         · in out String:
                                                                       Natural;
Boolean) is
    Temporary_Structure

begin

if To_The_Size = 0 then

The_String.The_Items := null;
elsif The_String.The_Items = null then

The_String.The_Items := new Substring(1 .. To_The_Size);
elsif To_The_Size > The_String.The_Items'Length then

if Preserve_The_Value then

Temporary_Structure := new Substring(1 ..

The_String.The_Length) :=
To The Size):
                       Temporary_Structure(1 .. The_String.The_Length) :=
   The_String.The_Items(1 .. The_String.The_Length);
The_String.The_Items := Temporary_Structure;
                        The String. The Items := new Substring(1 ...
To The Size):
           end if;
            The_String.The_Length := To_The_Size;
      end Set:
     begin
            Set (To The String,
            To_The_Size => From_The_String.The_Length,
Preserve_The_Value => False);
To_The_String.The_Items(1 .. From_The_String.The_Length) :=
From_The_String.The_Items(1 .. From_The_String.The_Length);
      exception
when Storage_Error =>
raise Overflow;
      end Copy;
     exception
            when Storage Error =>
      raise Overflow;
end Copy;
      To_The_Size => 0,
Preserve_The_Value => False);
      end Clear;
     New_Length)
            := To_The_String.The_Items(1 .. Old_Length);
To_The_String.The_Items(1 .. The_String.The_Length) :=
    The_String.The_Items(1 .. The_String.The_Length);
      exception
when Storage_Error =>
raise Overflow;
      begin
            Set (To The String,
```

```
To_The_Size => New_Length,
Preserve_The_Value => True);
To_The_String.The_Items((The_Substring'Length + 1) ...
New_Length)
          gtn)
:= To_The_String.The_Items(1 .. Old_Length);
To_The_String.The_Items(1 .. The_Substring'Length) :=
The_Substring;
      exception
          when Storage_Error => raise Overflow;
     end Prepend:
     New_Length : Natural :=
                            To_The_String.The_Length + The_String.The_Length;
     begin
     when Storage_Error =>
     raise Overflow;
end Append;
     New_Length : Natural :=
                            To_The_String.The_Length + The_Substring'Length;
     begin
           Set (To The String,
          To_The_Size => New_Length,
To_The_Size => True);
To_The_String.The_Items((Old_Length + 1) .. New_Length)
:= The_Substring;
      exception
           when Storage_Error => raise Overflow;
     end Append:
     The_String.The_Length;
End_Position : Natural :=
At_The_Position + The_String.The_Length;
           if At_The_Position > In_The_String.The_Length then
  raise Position_Error;
           else
                Set(In_The_String,
                1)) :=
                   The_String.The_Items(1 .. The_String.The_Length);
           end if
     exception
when Storage_Error =>
raise Overflow;
     procedure Insert (The_Substring : in Substring;
In_The_String : in out String;
At_The_Position : in Positive) is
Old_Length : Natural := In_The_String.The_Length;
New_Length : Natural := String.The_Length;
The_Substring'Length;
End_Position : Natural :=
At_The_Position + The_Substring'Length;
           if At_The_Position > In_The_String.The_Length then
  raise Position_Error;
                Set(In_The_String,
    To_The_Size => New_Length,
    Preserve_The_Value => True);
In_The_String.The_Items(End_Position .. New_Length) :=
    In_The_String.The_Items(At_The_Position .. Old_Length);
In_The_String.The_Items(At_The_Position .. (End_Position -.
           The_Substring; end if;
      exception
when Storage_Error =>
raise Overflow;
      end Insert:
      procedure Delete (In_The_String
                                                    : in out String:
```

```
: out Boolean) is
                                                                                                                                                        Result
                              From_The_Position : in
          To_The_Position
New_Length : Natural;
                                                                  Positive) is
                                                      : in
                                                                                                                                                          (Left, Right);
                                                                                                                   result := Is_Less_Than
end Is_Less_Than;
     begin
          if (From_The_Position > In_The_String.The_Length) or else
  (To_The_Position > In_The_String.The_Length) or else
  (From_The_Position > To_The_Position) then
  raise Position_Error;
                                                                                                                                                                                 : in String;
                                                                                                                   procedure Is_Greater_Than (Left
                                                                                                                                                                                 : in String;
: out Boolean) is
                                                                                                                                                        Result
                                                                                                                   result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
               e
New_Length := In_The_String.The_Length -
(To_The_Position - From_The_Position + 1);
In_The_String.The_Items(From_The_Position .. New_Length)
                                                                                                                                                                                 : in Substring;
: in String;
                                                                                                                   procedure Is_Greater_Than (Left
                                                                                                                                                        Right
                   In The String The Items
               : out Boolean) is
                                                                                                                                                        Result
                                                                                                                   begin
                                                                                                                   result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
          end if:
                                                                                                                                                                                 : in String;
: in Substring;
                                                                                                                   procedure Is_Greater_Than (Left
     end Delete:
                                                                                                                                                        Right
    Result
                                                                                                                                                                                 : out Boolean) is
                                                                                                                   result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
                                                                                                                                                                                 : in String;
: out Natural) is
                                                                                                                   procedure Length_Of
                                                                                                                                                       (The String
                                                                                                                                                        Result
     begin
          in
if (At_The_Position > In_The_String.The_Length) or else
    (End_Position > In_The_String.The_Length) then
                                                                                                                   begin
                                                                                                                   result := Length_Of
end Length_Of;
                                                                                                                                                            (The_String);
                raise Position_Error;
                                                                                                                                                                                 : in String;
: out Boolean) is
                                                                                                                                                       (The_String
                                                                                                                   procedure Is_Null
                In_The_String.The_Items(At_The_Position .. End_Position)
                                                                                                                                                        Result
                                                                                                                   result := Is_Null
end Is_Null;
                  With_The_String.The_Items(1 ...
                                                                                                                                                            (The_String);
With_The_String.The_Length);
     end if;
end Replace;
                                                                                                                                                                                 : in String;
: in Positive;
                                                                                                                                                       (The_String
                                                                                                                   procedure Item Of
                                                                                                                                                        At_The_Position
     procedure Replace (In_The_String : in out String;
At_The_Position : in Positive
With_The_Substring : in Substring
                                                                                                                                                                                 : out Item) is
                                                                     Positive;
Substring) is
                                                                                                                   begin
                                                                                                                                                            (The_String, At_The_Position);
                                                                                                                         result := Item Of
          End_Position : Natural :=
                                                                                                                   end Item_Of;
                               At_The_Position + With_The_Substring'Length -
1;
                                                                                                                                                                                 : in String;
: out Substring) is
                                                                                                                                                       (The_String
                                                                                                                   procedure Substring Of
     begin
   if (At_The_Position > In_The_String.The_Length) or else
      (End_Position > In_The_String.The_Length) then
                                                                                                                                                        Result
                                                                                                                   begin
                                                                                                                   result := Substring_Of
end Substring_Of;
                                                                                                                                                            (The_String);
                raise Position_Error;
          else
In_The_String.The_Items(At_The_Position .. End_Position)
                                                                                                                                                      (The_String : in String;
From_The_Position : in Positive;
To_The_Position : in Positive;
Result : out Substrin
                                                                                                                   procedure Substring_Of
                  With_The_Substring;
                                                                                                                                                                                 : out Substring) is
     end Replace;
                                                                                                                   begin
                                                                                                             result :=
Substring_Of(The_String,From_The_Position,To_The_Position);
end Substring_Of;
     procedure Set_Item (In_The_String : in out String;
At_The_Position : in Positive
With_The_Item : in Item) is
                                                                  Positive;
Item) is
          if At_The_Position > In_The_String.The_Length then
  raise Position_Error;
                                                                                                                   end of modification
          In_The_String.The_Items(At_The_Position) := With_The_Item; end if;
                                                                                                                   function Is_Equal (Left : in String; Right : in String) return Boolean is
                                                                                                                   begin
   if Left.The_Length /= Right.The_Length then
        return False;
     end Set Item:
     modified by Vincent Hong and Tuan Nguyen
date: 9 April 1995
adding procedures to replace functions
                                                                                                                              for Index in 1 .. Left.The_Length loop
if Left.The_Items(Index) /= Right.The_Items(Index)
                                                                                                             then
                                                                                                                                         return False:
                                                                                                                        retu
end if;
end loop;
return True;
end if;
Is p
                                                                   : in String;
     procedure Is_Equal
                                         (Left
                                                                    : in String:
                                                                     out Boolean) is
                                          Result
     result := Is_Equal
end Is_Equal;
                                              (Left, Right);
                                                                                                                   end Is_Equal;
                                                                                                                   function Is_Equal (Left : in Substring;
Right : in String) return Boolean is
                                                                     in Substring;
     procedure Is_Equal
                                         (Left
                                          Right
                                                                     in String;
                                                                                                                   begin
   if Left'Length /= Right.The_Length then
      return False;
                                                                    out Boolean) is
                                               (Left, Right);
     result := Is_Equal end Is_Equal;
                                                                                                                        else
for Index in 1 .. Left'Length loop
if Left(Left'First + Index - 1) /=
                                                                     in String;
in Substring;
     procedure Is Equal
                                         (Left
                                                                                                             if Lert(Lert'First
Right.The_Items(Index) then
return False;
end if;
end loop;
return True;
end if;
                                          Right
                                          Result
                                                                    : out Boolean) is
     result := Is_Equal end Is_Equal;
                                               (Left.Right):
                                                                                                                   end Is Equal;
                                                                     in String;
in String;
     procedure Is_Less_Than
                                         (Left
                                                                                                                   function Is_Equal (Left : in String;
Right : in Substring) return Boolean is
                                                                    : out Boolean) is
                                             (Left, Right);
     result := Is_Less_Than end Is_Less_Than;
                                                                                                                        if Left.The_Length /= Right'Length then
return False;
                                                                   : in Substring;
: in String;
: out Boolean) is
                                          (Left
     procedure Is_Less_Than
                                                                                                                              for Index in 1 .. Left.The_Length loop
if Left.The_Items(Index) /= Right(Right'First + Index
                                          Right
                                          Result
                                                                                                              - 1) then
                                                                                                                                   return False;
end if;
     begin
     result := Is_Less_Than end Is_Less_Than;
                                             (Left.Right);
                                                                                                                        end if;
end loop;
return True;
end if;
                                                                   : in String;
: in Substring;
     procedure Is_Less_Than
```

Right

```
end Is Equal:
        function Is_Less_Than (Left : in String;
Right : in String) return Boolean is
        begin
                for Index in 1 .. Left.The_Length loop
if Index > Right.The_Length then
return False;
elsif Left.The_Items(Index) < Right.The_Items(Index) then
                      return True;
elsif Right.The_Items(Index) < Left.The_Items(Index) then
                      return False;
end if;
               end loop;
return (Left.The_Length < Right.The_Length);
        end Is_Less_Than;
        function Is_Less_Than (Left : in Substring; Right : in String) return Boolean is
begin
   for Index in 1 .. Left'Length loop
      if Index > Right.The_Length then
            return False;
   elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
            return True;</pre>
                      elsif Right.The_Items(Index) < Left(Left'First + Index -
       return False;
end if;
end loop;
return (Left'Length < Right.The_Length);
end Is_Less_Than;
        function Is_Less_Than (Left : in String;
Right : in Substring) return Boolean is
              in
for Index in 1 .. Left.The_Length loop
    if Index > Right'Length then
        return False;
    elsif Left.The_Items(Index) < Right(Right'First + Index -</pre>
return True;
elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
return False;
end if;
end loop;
return (Left.The_Length < Right'Length);
end Is_Less_Than;
        function Is_Greater_Than (Left : in String;
Right : in String) return Boolean is
        begin
              in
for Index in 1 .. Left.The_Length loop
   if Index > Right.The_Length then
        return True;
   elsif Left.The_Items(Index) < Right.The_Items(Index) then</pre>
                      return False;
elsif Right.The_Items(Index) < Left.The_Items(Index) then
              return True;
end if;
end loop;
return False;
        end Is_Greater_Than;
        function Is_Greater_Than (Left : in Substring;
Right : in String) return Boolean is
       begin
for Index in 1 .. Left Length loop
if Index > Right The_Length then
```

```
elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
    return False;
    elsif Right.The_Items(Index) < Left(Left'First + Index -</pre>
1) then
                      return True:
     return T
end if;
end loop;
return False;
end Is_Greater_Than;
     function Is_Greater_Than (Left : in String; Right : in Substring) return Boolean is
     begin
           in
for Index in 1 .. Left.The_Length loop
   if Index > Right'Length then
        return True;
   elsif Left.The_Items(Index) < Right(Right'First + Index -</pre>
1) then
                      return False
return False;
elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
           return True;
end if;
end loop;
return False;
     end Is_Greater_Than;
     function Length_Of (The_String : in String) return Natural is
     return The_String.The_Length;
end Length_Of;
     function Is_Null (The_String : in String) return Boolean is
           return (The_String.The_Length = 0);
     end Is_Null;
     if At_The_Position > The_String.The_Length then
  raise Position_Error;
           return The_String.The_Items(At_The_Position); end if;
     end Item_Of;
     function Substring_Of (The_String : in String) return Substring is
    Temporary_Structure : Substring(1 .. 1);
     begin
          return The_String.The_Items(1 .. The_String.The_Length);
     exception
     when Constraint_Error =>
    return Temporary_Structure(1 .. 0);
end Substring_Of;
     Substring is
     itring is
begin
if (From_The_Position > The_String.The_Length) or else
    (To_The_Position > The_String.The_Length) or else
    (From_The_Position > To_The_Position) then
    raise Position_Error;
                return The String. The Items (From The Position ...
To_The_Position);
     end if;
end Substring_Of;
```

end String_Sequential_Unbounded_Unmanaged_Noniterator;

STRING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE String Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
GENERIC
      Ttem : PRIVATE TYPE,
   item : FRIVATE_TIME,
Substring : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive],
func_"e" : FUNCTION[Left : Item, Right : Item, RETURN : Boolean]
OPERATOR Copy
   SPECIFICATION
      INPUT
From_The_String : String,
To_The_String : String
OUTPUT
To_The_String : String
      EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR Copy
SPECIFICATION
      TNPIP
         From_The_Substring : Substring,
To_The_String : String
      OUTPUT
      To_The_String : String EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR Clear
   SPECIFICATION
INPUT
          The_String : String
       OUTPUT
          The_String : String
      EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR Prepend
   SPECIFICATION
INPUT
         NPUT
The_String : String,
To_The_String : String
      OUTPUT
To_The_String : String
EXCEPTIONS
          Overflow, Position_Error
    EMD
   OPERATOR Prepend
SPECIFICATION
       INPUT
The_Substring : Substring,
To_The_String : String
       OUTPUT
       To_The_String : String
EXCEPTIONS
         Overflow, Position_Error
    END
   OPERATOR Append
SPECIFICATION
       INPUT
The_String : String,
To_The_String : String
       To_The_String : String
EXCEPTIONS
Overflow, Position_Error
    END
    OPERATOR Append
SPECIFICATION
       INPUT
The_Substring : Substring,
To_The_String : String
       THISTHIO
       To_The_String : String
EXCEPTIONS
Overflow, Position_Error
    END
    OPERATOR Insert
    SPECIFICATION
      PECIFICATION
INPUT
The_String : String,
In_The_String : String,
At_The_Position : Positive
       OUTPUT
In_The_String : String
EXCEPTIONS
          Overflow, Position_Error
    END
    OPERATOR Insert
SPECIFICATION
```

The_Substring : Substring, In_The_String : String,

```
At_The_Position : Positive OUTPUT
   In_The_String : String
EXCEPTIONS
      Overflow, Position_Error
END
OPERATOR Delete
SPECIFICATION
   PECIFICATION
INPUT
In_The_String : String,
From_The_Position : Positive,
To_The_Position : Positive
   OUTPUT
      In_The_String : String
   EXCEPTIONS
Overflow, Position_Error
OPERATOR Replace
SPECIFICATION
   INPUT
In_The_String : String,
At_The_Position : Positive,
With_The_String : String
OUTPUT
In_The_String : String
   EXCEPTIONS
       Overflow, Position_Error
OPERATOR Replace
SPECIFICATION
   TNPITT
       In_The_String : String,
At_The_Position : Positive,
With_The_Substring : Substring
   OUTPUT
   In_The_String : String
EXCEPTIONS
       Overflow, Position_Error
END
OPERATOR Set_Item
SPECIFICATION
INFUT
In The_String: String,
At The Position: Positive,
With_The_Item: Item
OUTPUT
In. The_String: String
EXCEPTIONS
       Overflow, Position_Error
 OPERATOR Is Equal
 SPECIFICATION
   INPUT
Left : String,
Right : String
   OUTPUT
Result : Boolean
EXCEPTIONS
       Overflow, Position_Error
 END
 OPERATOR Is_Equal
SPECIFICATION
INPUT
Left: Substring,
Right: String
   OUTPUT
Result : Boolean
EXCEPTIONS
       Overflow, Position_Error
 OPERATOR Is_Equal
 SPECIFICATION
       Right : Substring
    OUTPUT
    Result : Boolean
EXCEPTIONS
       Overflow, Position_Error
 OPERATOR Is_Less_Than SPECIFICATION
    INPUT
Left : String,
Right : String
    Result : Boolean
EXCEPTIONS
Overflow, Position_Error
```

```
OPERATOR Is_Less_Than
SPECIFICATION
INPUT
Left : Substring,
Right : String
OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Less_Than SPECIFICATION
    INPUT
Left : String,
Right : Substring
    RIGHT: SAUSCIANG
OUTPUT
RESult: Boolean
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Greater_Than SPECIFICATION
    PECIFICATION
INPUT
Left: String,
Right: String
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Greater_Than SPECIFICATION
SPECIFICATION
INPUT
Left: Substring,
Right: String
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Greater_Than SPECIFICATION
    INPUT
Left : String,
Right : Substring
    OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Position_Error
END
OPERATOR Length_Of
```

```
SPECIFICATION
      INPUT
The_String : String
OUTPUT
Result : Natural
   EACEPTIONS
Overflow, Position_Error
END
  OPERATOR Is_Null
SPECIFICATION
INPUT
The_String: String
OUTPUT
Result: Boolean
EXCEPTIONS
Overflow, Position_Error
END
   OPERATOR Item_Of
SPECIFICATION
INPUT
      INPUT
The_String : String,
At_The_Position : Positive
OUTFUT
Result : Item
   EXCEPTIONS
Overflow, Position_Error
END
   OPERATOR Substring_Of
SPECIFICATION
INPUT
          The_String : String
      OUTPUT
Result : Substring
EXCEPTIONS
          Overflow, Position_Error
   END
   OPERATOR Substring_Of
  OPERATOR Substring_Of
SPECIFICATION
INPUT
The_String : String,
From_The_Position : Positive,
To_The_Position : Positive
OUTFUT
Result : Substring
EXCEPTIONS
          Overflow, Position_Error
   END
END
IMPLEMENTATION ADA String_Sequential_Unbounded_Unmanaged_Noniterator
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
procedure Length_Of
                                                                                                                                                                (The_String
                                                                                                                                                                                            : in String:
generic
                                                                                                                                                                                              out Natural);
in String;
                                                                                                                                                                 Result
       type Item is private:
                                                                                                                                                                (The String
type Item is private;
type Substring is array(Positive range <>) of Item;
with function "<" (Left : in Item;
Right : in Item) return Boolean;
package String_Sequential_Unbounded_Unmanaged_Iterator is
                                                                                                                          procedure Is_Null
                                                                                                                                                                                              out Boolean);
in String;
in Positive;
                                                                                                                                                                Result
(The_String
                                                                                                                          procedure Item_Of
                                                                                                                                                                 At_The_Position
                                                                                                                                                                Result
(The_String
                                                                                                                                                                                              out Item):
                                                                                                                                                                                              in String;
out Substring);
in String;
                                                                                                                          procedure Substring Of
      type String is limited private;
                                                                                                                                                                 Result
                                                                                                                                                                Result
(The_String
From_The_Position
To_The_Position
                                                              : in String,
: in out String);
: in Substring;
                                                                                                                          procedure Substring_Of
                                  (From_The_String
      procedure Copy
                                                                                                                                                                                              in Positive;
in Positive;
                                  To_The_String :
(From_The_Substring :
To_The_String :
(The_String :
      procedure Copy
                                                                                                                                                                                              out Substring);
                                                                                                                                                                 Result
                                                                 in out String);
in String;
      procedure Clear
                                                                                                                    -- end of modification
      procedure Prepend
                                  (The String
                                  To_The_String
(The_Substring
                                                                 in out String);
                                                                                                                                                               (Left
                                                                                                                                                                                          : in String;
: in String)
                                                                                                                          function Is_Equal
                                                                 in Substring;
in out String);
      procedure Prepend
                                                                                                                                                                                                                 return
                                                                                                                                                               Right
                                    To The String
                                                                                                                                                                                          : in Substring;
: in String
                                                                                                                    Boolean;
function Is_Equal
                                  (The_String
To_The_String
                                                                 in
                                                                            String:
      procedure Append
                                                                                                                                                               (Left
                                                                     out String);
Substring;
                                                                                                                                                                                                                 return
                                                                                                                                                               Right
      procedure Append
                                  (The_Substring
                                                                 in
                                                                                                                    Boolean;
function Is_Equal
                                  To_The_String
(The_String
In_The_String
                                                                 in out String);
                                                                                                                                                               (Left
                                                                                                                                                                                          : in String;
: in Substring) return
                                                                            String;
      procedure Insert
                                                                 in out String;
                                                                                                                                                                Right
                                                                                                                                                                                          : in String;
                                                                                                                    Boolean;
                                  At_The_Position
(The_Substring
                                                                 in
                                                                            Positive):
                                                                                                                          function Is_Less_Than
                                                                                                                                                               (Left
                                                                            Substring;
      procedure Insert
                                                                                                                                                                                                                  return
                                                                                                                                                                Right
                                                                 in out String;
in Positive);
                                    In The String
                                                                                                                    Boolean;
function Is_Less_Than
                                  At_The_String
At_The_Position
(In_The_String
From_The_Position
                                                                                                                                                                                          : in Substring;
: in String)
                                                                                                                                                               (Left
                                                                 in
                                                                           String;
Positive;
                                                                      out
      procedure Delete
                                                                                                                                                                                                                 return
                                                                                                                                                               Right
                                                                                                                    Boolean;
function Is_Less_Than
                                  To_The_Position
(In_The_String
At_The_Position
                                                                 in
                                                                            Positive);
                                                                                                                                                               (Left
                                                                                                                                                                                          : in String;
: in Substring) return
                                                                 in
in
                                                                     out String;
Positive;
      procedure Replace
                                                                                                                    Boolean;
                                                                            String);
                                   With The String
                                                                 in
                                                                                                                                                                                          : in String;
                                                                                                                                                              (Left
                                                                                                                          function Is_Greater_Than
                                  With_The_String
(In_The_String
At_The_Position
With_The_Substring
                                                                 in out
                                                                           String:
      procedure Replace
                                                                                                                                                                                           : in String)
                                                                                                                                                                                                                 return
                                                                            Positive:
                                                                  in
in
                                                                            Substring);
                                                                                                                    Boolean:
                                                                                                                          function Is_Greater_Than (Left
                                  (In_The_String
At_The_Position
With_The_Item
                                                                 in out String;
      procedure Set_Item
                                                                                                                                                                Right
                                                                                                                                                                                          : in String)
                                                                 in
                                                                            Positive:
                                                                                                                    Boolean:
                                                                                                                                                                                             in String;
                                                                                                                                                              (Left
                                                                                                                          function Is Greater Than
                                                                                                                                                                                          : in Substring) return
                                                                                                                                                                Right
                                                                                                                    Boolean;
      modified by Vincent Hong and Tuan Nguyen
                                                                                                                                                                                          : in String)
                                                                                                                                                                                                                 return
                                                                                                                          function Length Of
                                                                                                                                                               (The String
      date: 9 April 1995
adding procedures to replace functions
                                                                                                                    Natural;
function Is_Null
                                                                                                                                                                                          : in String)
                                                                                                                                                                                                                 return
                                                                                                                                                               (The_String
                                                                          in String;
in String;
                                                                                                                    Boolean:
      procedure Is_Equal
                                            (Left
                                                                                                                                                              (The_String
At_The_Position
                                                                                                                                                                                          : in String;
                                                                                                                          function Item_Of
                                            Right
Result
(Left
                                                                         in String;
out Boolean);
in Substring;
in String;
out Boolean);
in String;
out Boolean);
in String;
in String;
                                                                                                                                                                                          : in Positive)
                                                                                                                                                                                                               return
      procedure Is_Equal
                                                                                                                                                               (The String
                                                                                                                                                                                          : in String) return
                                                                                                                          function Substring_Of
                                             Right
                                                                                                                    Substring;
function Substring_Of
                                             Result
(Left
                                                                                                                                                              (The_String : in String;
From_The_Position : in Positive;
To_The_Position : in Positive)return
      procedure Is Equal
                                             Right
                                             Result
                                                                                                                    Substring;
      procedure Is_Less_Than
                                             /Left
                                             Right
Result
                                                                           out Boolean);
in Substring;
                                                                                                                          generic
                                                                                                                                with procedure Process (The_Item : in Item;
Continue : out Boolean);
      procedure Is_Less Than
                                             (Left
                                                                          in Substring;
out Boolean);
in String;
in Substring;
out Boolean);
in String;
                                             Right
Result
(Left
Right
                                                                                                                          procedure Iterate (Over_The_String : in String);
                                                                                                                          Overflow : exception;
Position_Error : exception;
      procedure Is_Less_Than
                                              Result
                                            (Left
      procedure Is_Greater_Than
                                                                          in String;
                                                                                                                    private
                                              Right
                                                                                                                          ate
type Structure __
type String is
record
The_Length : Natural := 0;
The Items : Structure;
                                                                                                                           type Structure is access Substring;
                                                                          out Boolean);
in Substring;
                                              Result
      procedure Is_Greater_Than (Left
                                             Right
Result
                                                                          in String;
                                                                          out Boolean);
in String;
in Substring;
      procedure Is_Greater_Than (Left
                                                                                                                    end record;
end String_Sequential_Unbounded_Unmanaged_Iterator;
                                             Right
Result
                                                                          out Boolean):
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
package body String_Sequential_Unbounded_Unmanaged_Iterator is
      procedure Set (The_String : in out String;
To_The_Size : in Natural
Preserve_The_Value : in Boolean
Temporary_Structure : Structure;
                                                                                 Boolean) is
      To_The_Size);
                           Temporary_Structure(1 ... The_String.The_Length) :=
   The_String.The_Items(1 ... The_String.The_Length);
The_String.The_Items := Temporary_Structure;
                     else
   The_String.The_Items := new Substring(1 ...
To_The_Size);
              end if;
              The_String.The_Length := To_The_Size;
      To_The_String
       begin
              Set(To_The_String,
                 tc(10_ine_String, => From_The_String.The_Length,
    To_The_Size => False);
    The_String.The_Items(1 . . From_The_String.The_Length) :=
    From_The_String.The_Items(1 . . From_The_String.The_Length);
              when Storage_Error => raise Overflow;
       end Copy:
      procedure Copy (From_The_Substring : in Substring; To_The_String : in out String) is
              In
Set(To_The_String,
    To_The_Size => From_The_Substring'Length,
    Freserve_The_Value => False);
To_The_String.The_Items(1 .. From_The_Substring'Length) :=
    From_The_Substring;
       exception
when Storage_Error =>
raise Overflow;
       end Copy;
       procedure Clear (The_String : in out String) is
              Set (The String,
                    To_The_Size => 0,
Preserve_The_Value => False);
       end Clear:
      begin
              To_The_String, => New_Length,
Preserve_The_Value => True);
To_The_String.The_Items((The_String.The_Length + 1) ...
New Length)
              yth)
    := To_The_String.The_Items(1 .. Old_Length);
    To_The_String.The_Items(1 .. The_String.The_Length) :=
        The_String.The_Items(1 .. The_String.The_Length);
       exception
              when Storage_Error => raise Overflow;
       end Prepend:
       procedure Prepend (The_Substring : in Substring;
To_The_String : in out String) is
Old_Length : Natural := To_The_String.The_Length;
New_Length : Natural :=
                                     To_The_String.The_Length + The_Substring'Length;
       begin
              Set (To_The_String,
```

```
To_The_Size => New_Length,
Preserve_The_Value => True);
To_The_String.The_Items((The_Substring'Length + 1) ...
New Length)
                       := To_The_String.The_Items(1 .. Old_Length);
To_The_String.The_Items(1 .. The_Substring'Length) :=
The Substring;
             exception
when Storage_Error =>
                                  raise Overflow;
             end Prepend:
          exception
                         when Storage Error =>
                                   raise Overflow;
             end Append;
           procedure Append (The_Substring : in Substring;
To_The_String : in out String) is
Old_Length : Natural := To_The_String.The_Length;
New_Length : Natural :=
To_The_String.The_Length;
To_The_String.The_Versity | The Versity | Th
                                                              To_The_String.The_Length + The_Substring'Length;
            begin
                        Set (To The String,
                        TO_The_Size => New_Length,
    Freserve_The_Value => True);
TO_The_String.The_Items((Old_Length + 1) .. New_Length)
    := The_Substring;
             exception
                         when Storage_Error => raise Overflow;
            end Append;
           procedure Insert (The_String : in String;
In_The_String : in out String;
At_The_Position : in Positive) is
                                                                   Natural := In_The_String.The_Length;
Natural :=
                        Old Length
                        New_Length
The_String.The_Length;
End_Position : Natural :=
At_The_Position + The_String.The_Length;
                                                                   In_The_String.The_Length +
                       if At_The_Position > In_The_String.The_Length then
  raise Position_Error;
                         else
                                  e
Set(In_The_String,
To_The_Size => New_Length,
Preserve_The_Value => True);
In_The_String.The_Items(End_Position .. New_Length) :=
In_The_String.The_Items(At_The_Position .. Old_Length);
In_The_String.The_Items(At_The_Position .. (End_Position -
                                          The_String.The_Items(1 .. The_String.The_Length);
                       end if:
           end 1.,
exception
when Storage_Error =>
raise Overflow;
           procedure Insert (The_Substring : in Substring;
In_The_String : in out String;
At_The_Position : in Positive) is
                                                                   Natural := In_The_String.The_Length;
Natural :=
In_The_String.The_Length +
                        New_Length
The Substring'Length:
                        End_Position : Natural := At_The_Position + The_Substring'Length;
                        if At_The_Position > In_The_String.The_Length then
raise Position_Error;
                       else
Set(In_The_String,
                                   Set(In_Ine_String,

To_The_Size => New_Length,

Preserve_The_Value => True);

In_The_String.The_Items(End_Position .. New_Length) :=

In_The_String.The_Items(At_The_Position .. Old_Length);

In_The_String.The_Items(At_The_Position .. (End_Position -
                                          The_Substring;
                        end if:
            exception =>
when Storage_Error =>
raise Overflow;
             procedure Delete (In The String
                                                                                                                        : in out String;
```

```
: out Boolean) is
                                                                                                                                                          Result
                                                                    Positive;
                              From The Position : in
          To_The_Position : in
New_Length : Natural;
                                                                   Positive) is
                                                                                                                                                            (Left, Right);
                                                                                                                     result := Is_Less_Than end Is_Less_Than;
     begin
           .n
if (From_The_Position > In_The_String.The_Length) or else
(To_The_Position > In_The_String.The_Length) or else
(From_The_Position > To_The_Position) then
                                                                                                                                                                                    : in String;
                                                                                                                     procedure Is_Greater_Than (Left
                                                                                                                                                                                    : in String;
: out Boolean) is
                                                                                                                                                           Right
                                                                                                                                                          Result
                raise Position_Error;
                                                                                                                     result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
                · in Substring:
                                                                                                                     procedure Is_Greater_Than (Left
                In_The_String.The_Items
  ((To_The_Position + 1) ... In_The_String.The_Length);
Set(In_The_String,
   To_The_Size => New_Length,
   Preserve_The_Value => True);
                                                                                                                                                                                    : in String;
: out Boolean) is
                                                                                                                                                          Right
                                                                                                                                                          Result
                                                                                                                     begin
                                                                                                                          result := Is_Greater_Than (Left,Right);
                                                                                                                     end Is_Greater_Than;
           end if:
                                                                                                                                                                                    : in String;
: in Substring;
                                                                                                                     procedure Is_Greater_Than (Left
     end Delete:
                                                                                                                                                           Right
                                                                                                                                                                                    : out Boolean) is
    procedure Replace (In_The_String : in out String;
At_The_Position : in Positive;
With_The_String : in String) is
End_Position : Natural :=
At_The_Position + With_The_String.The_Length -
                                                                                                                                                          Result
                                                                                                                     begin
                                                                                                                     result := Is_Greater_Than (Left,Right);
end Is_Greater_Than;
                                                                                                                                                                                    : in String;
: out Natural) is
                                                                                                                     procedure Length_Of
                                                                                                                                                         (The_String
                                                                                                                                                           Result
     begin
                                                                                                                     begin
          in
if (At_The_Position > In_The_String.The_Length) or else
    (End Position > In_The_String.The_Length) then
                                                                                                                     result := Length_Of
end Length_Of;
                                                                                                                                                               (The_String);
                raise Position_Error;
                                                                                                                                                                                    : in String;
                                                                                                                                                         (The_String
                In_The_String.The_Items(At_The_Position .. End_Position)
                                                                                                                     procedure Is_Null
                                                                                                                                                                                    : out Boolean) is
                                                                                                                                                          Result
                   With_The_String.The_Items(1 ...
                                                                                                                          result := Is Null
                                                                                                                                                               (The_String);
With_The_String.The_Length);
                                                                                                                     end Is_Null;
     end if;
end Replace;
                                                                                                                                                                                   : in String;
: in Positive;
                                                                                                                                                         (The_String
                                                                                                                     procedure Item_Of
                                                                                                                                                          At_The_Position
Result
    procedure Replace (In_The_String : in out String;
At_The_Position : in Positive;
With_The_Substring : in Substring) is
End_Position : Natural :=
At_The_Position + With_The_Substring'Length -
                                                                                                                                                                                    : out Item) is
                                                                                                                                                               (The_String, At_The_Position);
                                                                                                                     result := Item_Of
end Item_Of;
                                                                                                                     procedure Substring_Of
                                                                                                                                                         (The String
                                                                                                                                                                                    : in String;
                                                                                                                                                                                    : out Substring) is
     begin
           if (At_The_Position > In_The_String.The_Length) or else
    (End_Position > In_The_String.The_Length) then
    raise Position_Error;
                                                                                                                     begin
                                                                                                                     result := Substring_Of
end Substring_Of;
                                                                                                                                                               (The String);
           else
                In_The_String.The_Items(At_The_Position .. End_Position)
                                                                                                                                                         (The_String : in String;
From_The_Position : in Positive;
To_The_Position : in Positive;
Result : out Substring) is
                                                                                                                     procedure Substring_Of
                   With_The_Substring;
     end Replace;
                                                                                                                     begin
                                                                                                                          result :=
     procedure Set_Item (In_The_String : in out String;
At_The_Position : in Positiv
With_The_Item : in Item) i
                                                                                                               result :=
Substring_Of(The_String,From_The_Position,To_The_Position);
end Substring_Of;
                                                                    Positive:
                                                                   Ttem) is
           if At_The_Position > In_The_String.The_Length then
  raise Position_Error;
                                                                                                                -- end of modification
                                                                                                                     function Is_Equal (Left : in String; Right : in String) return Boolean is
           else
                In_The_String.The_Items(At_The_Position) := With_The_Item;
           end if;
                                                                                                                          if Left.The_Length /= Right.The_Length then
    return False;
     end Set Item;
                                                                                                                          else
                                                                                                                                for Index in 1 .. Left.The_Length loop if Left.The_Items(Index) /= Right.The_Items(Index)
     modified by Vincent Hong and Tuan Nguyen date: 9 April 1995
     adding procedures to replace functions
                                                                                                               then
                                                                                                                                           return False:
                                                                    : in String;
                                                                                                                               end if;
end loop;
return True;
     procedure Is_Equal
                                                                    : in String;
: out Boolean) is
                                           Result
                                                                                                                           end if;
                                               (Left.Right);
           result := Is_Equal
                                                                                                                     end Is_Equal;
     end Is_Equal;
                                                                                                                     function Is_Equal (Left : in Substring;
Right : in String) return Boolean is
                                                                     : in Substring;
                                          (Left
     procedure Is_Equal
                                                                     : in String;
: out Boolean) is
                                           Result
                                                                                                                           if Left'Length /= Right.The_Length then
                                                                                                                                return False;
                                               (Left.Right);
           result := Is_Equal
                                                                                                               else
    for Index in 1 .. Left'Length loop
    if Left(Left'First + Index - 1) /=
Right.The_Items(Index) then
    return False;
    end if;
     end Is_Equal;
     procedure Is_Equal
                                          (Left
                                                                     : in String;
                                                                      in Substring;
                                           Result
                                                                     : out Boolean) is
                                                                                                                           end loop;
return True;
end if;
                                               (Left.Right);
           result := Is_Equal
     end Is_Equal;
                                                                                                                     end Is_Equal;
                                                                     : in String;
: in String;
                                          (Left
     procedure Is_Less_Than
                                                                                                                     function Is_Equal (Left : in String;
Right : in Substring) return Boolean is
                                                                     : out Boolean) is
                                                                                                                           if Left.The_Length /= Right'Length then return False;
     result := Is_Less_Than end Is_Less_Than;
                                             (Left.Right);
                                                                     : in Substring;
                                           (Left
     procedure Is_Less_Than
                                                                                                                                for Index in 1 .. Left.The_Length loop
if Left.The_Items(Index) /= Right(Right'First + Index
                                           Right
Result
                                                                     : in String;
                                                                     · out Boolean) is
                                                                                                                                      return False;
end if;
                                               (Left, Right);
      result := Is_Less_Than end Is_Less_Than;
                                                                                                                           end loop;
return True;
end if;
                                                                    : in String;
: in Substring;
                                           (Left
     procedure Is_Less_Than
```

Right

```
end Is_Equal;
      function Is_Less_Than (Left : in String;
Right : in String) return Boolean is
            for Index in 1 .. Left.The_Length loop
if Index > Right.The_Length then
return False;
elsif Left.The_Items(Index) < Right.The_Items(Index) then
                  return True;
elsif Right.The_Items(Index) < Left.The_Items(Index) then
                  return False;
end if;
            end if;
end loop;
return (Left.The_Length < Right.The_Length);</pre>
      end Is_Less_Than;
      function Is_Less_Than (Left : in Substring;
Right : in String) return Boolean is
            for Index in 1 .. Left'Length loop
if Index > Right.The_Length then
                  return False;
elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
return True;
elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
                        return False:
      return False;
end if;
end loop;
return (Left'Length < Right.The_Length);
end Is_Less_Than;
      function Is_Less_Than (Left : in String; Right : in Substring) return Boolean is
      begin
            for Index in 1 .. Left.The_Length loop if Index > Right'Length then
                  return False;
elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
                        return True:
return frue;
elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
return False;
end if;
      end loop;
enturn (Left.The_Length < Right'Length);
end Is_Less_Than;
      begin
            in
for Index in 1 .. Left.The_Length loop
   if Index > Right.The_Length then
        return True;
   elsif Left.The_Items(Index) < Right.The_Items(Index) then</pre>
                  return False; elsif Right.The_Items(Index) < Left.The_Items(Index) then return True; end if;
      end loop;
return False;
end Is_Greater_Than;
      begin
begin
   for Index in 1 .. Left'Length loop
    if Index > Right.The_Length then
        return True;
   elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
        return False;</pre>
                  elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
                        return True:
```

```
end if;
end loop;
return False;
     end Is_Greater_Than;
     function Is_Greater_Than (Left : in String;
Right : in Substring) return Boolean is
          for Index in 1 .. Left.The_Length loop
if Index > Right'Length then
               return True;
elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
return False;
elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
               return True;
end if;
          end loop;
return False;
     end Is Greater Than:
     function Length_Of (The_String : in String) return Natural is
     return The_String.The_Length;
end Length_Of;
     function Is_Null (The_String : in String) return Boolean is
     return (The_String.The_Length = 0);
end Is_Null;
     if At_The_Position > The_String.The_Length then
               raise Position_Error;
          return The_String.The_Items(At_The_Position);
end if;
     end Item_Of;
     function Substring_Of (The_String : in String) return Substring is
   Temporary_Structure : Substring(1 .. 1);
     begin
          return The_String.The_Items(1 .. The_String.The_Length);
     exception
when Constraint_Error =>
     return Temporary_Structure(1 .. 0); end Substring_Of;
    function Substring_Of (The_String : in String,
From_The_Position : in Positive;
To The_Position : in Positive) return
Substring is
     begin
          In if (From_The_Position > The_String.The_Length) or else (To_The_Position > The_String.The_Length) or else (From_The_Position > To_The_Position) then
               raise Position_Error;
else
return The_String.The_Items(From_The_Position ..
To_The_Position);
          else
     end if;
end Substring_Of;
     procedure Iterate (Over_The_String : in String) is
          Continue : Boolean;
    begin
for The_Iterator in 1 .. Over_The_String.The_Length loop
Process(Over_The_String.The_Items(The_Iterator),
Continue):
     exit when not Continue;
end loop;
end Iterate;
end String_Sequential_Unbounded_Unmanaged_Iterator;
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE String_Sequential_Unbounded_Unmanaged_Iterator SPECIFICATION
  substring : RRRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive],
func_"<" : FUNCTION[Left : Item, Right : Item, RETURN : Boolean]
OPERATOR Copy
SPECIFICATION
TANDIUM</pre>
      INPUT
      From The String : String,
To The String : String
OUTPUT
To The String : String
      EXCEPTIONS
         Overflow, Position_Error
   OPERATOR Copy
SPECIFICATION
      TNPUT
         From_The_Substring : Substring,
To_The_String : String
      OUTPUT
      To_The_String : String
EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR Clear
   SPECIFICATION
         The_String : String
      OUTPUT
         The_String : String
      EXCEPTIONS
Overflow, Position_Error
   END
   OPERATOR Prepend
SPECIFICATION
      INPUT
      INPUT
The_String : String,
TO_The_String : String
OUTPUT
TO_The_String : String
      EXCEPTIONS
         Overflow, Position_Error
   OPERATOR Prepend
   SPECIFICATION
INPUT
      INPUT
The_Substring : Substring,
To_The_String : String
OUTPUT
To_The_String : String
      EXCEPTIONS
         Overflow, Position_Error
   OPERATOR Append
SPECIFICATION
      INPUT
      The_String : String,
To_The_String : String
OUTPUT
To_The_String : String
      EXCEPTIONS
Overflow, Position_Error
   END
   OPERATOR Append
SPECIFICATION
      INPUT
         The_Substring : Substring,
To_The_String : String
         To The String : String
      EXCEPTIONS
Overflow, Position_Error
   OPPRATOR Insert
    SPECIFICATION
INPUT
         IPUT
The_String : String,
In_The_String : String,
At_The_Position : Positive
      OUTPUT
In_The_String : String
       EXCEPTIONS
         Overflow, Position_Error
   END
   OPERATOR Insert
SPECIFICATION
       TNPITT
```

The_Substring : Substring, In_The_String : String,

```
At_The_Position : Positive
   OUTPUT
In_The_String : String
   EXCEPTIONS
      Overflow, Position_Error
END
OPERATOR Delete
SPECIFICATION
   INPUT
      NPUT
In_The_String : String,
From_The_Position : Positive,
To_The_Position : Positive
   OUTPUT
   In_The_String : String
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Replace
SPECIFICATION
   INPUT
In_The_String : String,
At_The_Position : Positive,
With_The_String : String
   OUTPUT
       In_The_String : String
   EXCEPTIONS
      Overflow, Position_Error
OPERATOR Replace SPECIFICATION
   INPUT
      NFOT
In_The_String : String,
At_The_Position : Positive,
With_The_Substring : Substring
   OUTPUT
      In The String : String
   EXCEPTIONS
Overflow, Position_Error
OPERATOR Set_Item
SPECIFICATION
INPUT
In_The_String: String,
At_The_Position: Positive,
With_The_Item: Item
OUTPUT
In_The_String: String
   In_The_String : String
EXCEPTIONS
Overflow, Fosition_Error
OPERATOR Is_Equal SPECIFICATION
   TNPITT
      Left : String,
Right : String
   OUTPUT
       Result : Boolean
   EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Equal SPECIFICATION
   INPUT
Left : Substring,
       Right : String
   OUTTRUTT
   Result : Boolean
EXCEPTIONS
      Overflow, Position_Error
END
OPERATOR Is_Equal SPECIFICATION
    INPUT
      Left : String,
Right : Substring
       Result : Boolean
   EXCEPTIONS
Overflow, Position_Error
 END
OPERATOR Is_Less_Than SPECIFICATION
   INPUT
Left : String,
   OUTPUT
       Result : Boolean
    EXCEPTIONS
       Overflow, Position_Error
 END
```

```
OPERATOR Is_Less_Than SPECIFICATION
    INFUT
Left : Substring,
Right : String
    OUTPUT
Result : Boolean
EXCEPTIONS
Overflow, Position_Error
END
OPERATOR Is_Less_Than
SPECIFICATION
INPUT
Left : String,
Right : Substring
    OUTPUT
Result : Boolean
EXCEPTIONS
        Overflow, Position_Error
OPERATOR Is_Greater_Than
 SPECIFICATION
INPUT
Left: String,
    Right : String
OUTPUT
Result : Boolean
EXCEPTIONS
         Overflow, Position_Error
OPERATOR Is Greater_Than
SPECIFICATION
INPUT
Left: Substring,
Right: String
OUTPUT
Result: Boolean
EXCEPTIONS
        Overflow, Position_Error
OPERATOR Is Greater_Than
SPECIFICATION
INPUT
Left: String,
Right: Substring
OUTPUT
Result: Boolean
EXCEPTIONS
        Overflow, Position_Error
 END
OPERATOR Length_Of
SPECIFICATION
INPUT
The_String : String
     OUTPUT
        Result : Natural
```

```
Overflow, Position_Error
  OPERATOR Is_Null SPECIFICATION
     INPUT
        The_String : String
     OUTPUT
Result : Boolean
EXCEPTIONS
  Overflow, Position_Error
  OPERATOR Item_Of
SPECIFICATION
INPUT
     INPUT
The_String : String,
At_The_Position : Positive
OUTPUT
Result : Item
EXCEPTIONS
  EXCEPTIONS
Overflow, Position_Error
END
  OPERATOR Substring_Of
SPECIFICATION
INPUT
     INPUT
The_String : String
OUTPUT
Result : Substring
EXCEPTIONS
        Overflow, Position_Error
  END
  OPERATOR Substring_Of
  SPECIFICATION
INPUT
     INPUT
The_String : String,
From The_Position : Positive,
To_The_Position : Positive
OUTPUT
Result : Substring
EXCEPTIONS
        Overflow, Position_Error
  OPERATOR Iterate
  SPECIFICATION
GENERIC
        Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
     Over_The_String : String
EXCEPTIONS
        Overflow, Position_Error
  END
IMPLEMENTATION ADA String_Sequential_Unbounded_Unmanaged_Iterator END
```

TREE ARBITRARY DOUBLE UNBOUNDED MANAGED

```
generic
generic
type Item is private;
Expected Number_Of_Children : in Positive;
package Tree_Arbitrary_Double_Unbounded_Managed is
     type Tree is private;
     Null_Tree : constant Tree;
    modified by Tuan Nguyen
25 December 1995
adding procedures to replace functions
                                                  (Teft
                                                                  · in Tree:
     procedure Is_Equal
                                                                 : in Tree;
: in Tree;
: out Boolean);
: in Tree;
: out Boolean);
: in Tree);
                                                   Right
                                                   Result
                                                  (The_Tree
Result
(The_Tree
     procedure Is_Null
     procedure Item_Of
                                                                  : out Item):
```

```
procedure Number_Of_Children_In (The_Tree : in Tree;
                                                      Result : out Natural);
(The_Tree : in Tree;
The_Child : in Positive;
Result : out Tree);
      procedure Child_Of
-- end of modification
                                                                     : in Tree;
: in Tree)
      function Is_Equal
                                                     (Left
                                                                                            return
Boolean:
function Is_Null
Boolean;
function Item_Of
                                                     (The_Tree : in Tree)
                                                     (The_Tree : in Tree)
                                                                                            return
Item;
      function Number_Of_Children_In (The_Tree : in Tree)
Natural;
function Child_Of
                                                     (The_Tree : in Tree;
The_Child : in Positive) return
Tree; function Parent_Of
                                                     (The_Tree : in Tree)
                                                                                            return
Tree;
                               : exception;
      Overflow
      Overriow : exception;
Tree_Is_Null : exception;
Tree_Is_Not_Null : exception;
Not_At_Root : exception;
                           : exception;
: exception;
      Child_Error
private
type Node;
  type Tree is access Node;
  Null_Tree : constant Tree := null;
end Tree_Arbitrary_Double_Unbounded_Managed;
```

TREE ARRITRARY DOUBLE UNBOUNDED MANAGED

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
- restrictions as set forth in Subdivision (3) (3) - of the rights in Technical Data and Computer - Software Clause of FAR 52.227-7013. Manufacturer: - Wizard software, 2171 S. Parfet Court, Lakewood, - Colorado 80227 (1-303-987-1874)
with Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator,
Storage_Manager_Sequential;
package body Tree_Arbitrary_Double_Unbounded_Managed is
      function Hash_Of (The_Child : in Positive) return Positive;
     package Children is new
        lckage Children is new
Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
(Domain => Positive,
   Ranges => Tree,
   Number_Of_Buckets => Expected_Number_Of_Children,
   Hash_Of => Hash_Of);
     type Node is
           record
Previous
                                  : Tree;
: Item;
                The_Item : Item;
The_Children : Children.Map;
Next : Tree;
      function Hash_Of (The_Child : in Positive) return Positive is
           return The_Child;
     end Hash_Of;
     procedure Free (The_Node : in out Node) is begin
           The_Node.Previous := null;
Children.Clear(The_Node.The_Children);
      end Free;
     The_Node.Next := To_Next;
      function Next_Of (The_Node : in Node) return Tree is
     return The_Node.Next;
end Next_Of;
     => Node,
=> Tree,
=> Free,
                                                   Free => Free,
Set_Pointer => Set_Next,
Pointer_Of => Next_Of);
     Temporary_Node : Tree;
           begin
                end if;
Continue := True;
end Copy_Child;
           procedure Copy_Children is new Children.Iterate(Copy_Child);
     begin
           in
Clear(To_The_Tree);
if From_The_Tree /= null then
    To_The_Tree := Node_Manager.New_Item;
    To_The_Tree.The_Item := From_The_Tree.The_Item;
    Copy_Children(From_The_Tree.The_Children);
end if;
      exception
           when Storage_Error | Children.Overflow => raise Overflow;
      end Copy;
     Temporary_Node : Tree := The_Range;
```

```
Clear(Temporary_Node);
Continue := True;
                Continue := end Clear_Child;
                procedure Clear_Children is new Children.Iterate(Clear_Child);
                if The_Tree /= null then
   Clear_Children(The_Tree.The_Children);
                       Node_Manager.Free(The_Tree);
         end Clear:
        procedure Construct (The_Item : in Item;
And_The_Tree : in out Tree;
Number_Of_Children : in Nature
                                                                                                     Natural
                On_The_Child
Temporary_Node : Tree;
                                                                                     : in
                                                                                                     Natural) is
        begin
               in
if Number_Of_Children = 0 then
    if And_The_Tree = null then
    And_The_Tree := Node_Manager.New_Item;
    And_The_Tree.The_Item := The_Item;
                             return:
               else
raise Tree_Is_Not_Null;
end if;
elsif On_The_Child > Number_Of_Children then
raise Child_Error;
elsif And_The_Tree = null then
And_The_Tree := Node_Manager.New_Item;
And_The_Tree.The_Item := The_Item;
for Index in 1 . Number_Of_Children loop
Children_Bind(The_Domain => Index,
And_The_Range => null,
In_The_Map =>
                        else
                                                         In_The_Map
 And_The_Tree.The_Children);
end loop;
elsif And_The_Tree.Previous = null then
                       Children.Bind
                                           (The_Domain => Index,
And_The_Range => null,
In_The_Map => Temporary_Node.The_Children);
                               end if:
                       end loop;
and The_Tree.Previous := Temporary_Node;
And_The_Tree := Temporary_Node;
                else
                raise Not_At_Root;
end if;
         exception
         when Storage_Error | Children.Overflow => raise Overflow; end Construct;
        begin
   Of_The_Tree.The_Item := To_The_Item;
         exception
                when Constraint Error =>
         raise Tree_Is_Null;
end Set_Item;
        procedure Swap_Child (The_Child : in Positive Of_The_Tree : in out Tree;
And_The_Tree : in out Tree) is
                                                                                            Positive:
               Temporary_Node : Tree;
Of_The_Tree.The_Children);

Children.Unbind(The_Child, Of_The_Tree.The_Children);
Children.Bind(The_Domain => The_Child,
And_The_Range => null,
In_The_Map => Of_The_Tree.The_Children);

if Temporary_Node /= null then
Temporary_Node.Previous := null;
end if;
And_The_Tree := Temporary_Node;
elsif And_The_Tree.Previous = null then
Temporary_Node := Children.Range_Of
(The_Domain => The_Child,
In_The_Map =>

Of_The_Tree_The_Children);
 Of_The_Tree.The_Children);
Children.Unbind(The_Child, Of_The_Tree.The_Children);
Children.Bind(The_Domain => The_Child,
And_The_Range => And_The_Tree,
In_The_Map => Of_The_Tree.The_Children);
```

```
raise Not_At_Root;
end if;
    end if;
exception
   when Constraint_Error =>
        raise Tree_Is_Null;
   when Children.Domain_Is_Not_Bound =>
        raise Child_Error;
end Swap_Child;
   modified by Tuan Nguyen
25 December 1995
    adding procedures to replace functions
                                          (Left
                                                      : in Tree;
: in Tree;
: out Boolean) is
    procedure Is_Equal
                                           Right
                                           Result
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;
                                          (The_Tree : in Tree;
Result : out Boolean) is
    procedure Is_Null
         Result := Is_Null(The_Tree);
    end Is_Null;
                                          (The_Tree : in Tree);
Result : out Item) is
    procedure Item_Of
    begin
        Result := Item_Of(The_Tree);
    end Item_Of;
    Result := Number_Of_Children_In(The_Tree);
end Number_Of_Children_In;
                                          (The_Tree : in Tree;
The_Child : in Positive;
Result : out Tree) is
    procedure Child_Of
         Result := Child_Of(The_Tree,The_Child);
    end Child_Of;
    end of modification
             in
if not Is_Equal(The_Range,
Children.Range_Of(The_Domain,
Right.The_Children))
then
                  Trees_Are_Equal := False;
Continue := False;
```

```
else
Continue := True;
end if;
end check_Child_Equality;
procedure Check_Equality is new
Children.Iterate(Check_Child_Equality);
    begin
if Left.The_Item /= Right.The_Item then
return False;
         else
if Children.Extent_Of(Left.The_Children) /=
Children.Extent_Of(Right.The_Children) then
        else
Check_Equality(Left.The_Children);
return Trees_Are_Equal;
end if;
end if;
eption
    exception
when Constraint_Error =>
return (Left = Null_Tree) and (Right = Null_Tree);
    end Is Equal;
    function Is_Null (The_Tree : in Tree) return Boolean is
    return (The_Tree = null);
end Is_Null;
    function Item_Of (The_Tree : in Tree) return Item is
    begin
        return The_Tree.The_Item;
    exception when Constraint_Error =>
             raise Tree_Is_Null;
    end Item_Of;
    function Number_Of_Children_In (The_Tree : in Tree) return Natural
    begin
         return Children.Extent_Of(The_Tree.The_Children);
    exception when Constraint_Error =>
    raise Tree_Is_Null;
end Number_Of_Children_In;
    raise Child_Error;
end Child_Of;
     function Parent_Of (The_Tree : in Tree) return Tree is
    begin
return The_Tree.Previous;
    exception when Constraint_Error =>
    raise Tree_Is_Null;
end Parent_Of;
end Tree_Arbitrary_Double_Unbounded_Managed;
```

TREE ARBITRARY DOUBLE UNBOUNDED MANAGED

PSDL

```
TYPE Tree_Arbitrary_Double_Unbounded_Managed
SPECIFICATION
GENERIC
   GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
INFUT
From_The_Tree : Tree,
To_The_Tree : Tree
OUTPUT
To_The_Tree : Tree
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
   OPERATOR Clear
   SPECIFICATION
      INPUT
The_Tree : Tree
       OUTPUT
      The_Tree : Tree
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
   OPERATOR Construct
   SPECIFICATION
      PECIFICATION
INPUT
The_Item : Item,
And_The_Tree : Tree,
Number_Of_Children : Natural,
On_The_Child : Natural
OUTPUT
And_The_Tree : Tree
      EXCEPTIONS
          Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
   END
   OPERATOR Set_Item SPECIFICATION
      INPUT
      Of_The_Tree : Tree,
To_The_Item : Item
OUTPUT
      OUTPUT
Of_The_Tree : Tree
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
   OPERATOR Swap_Child
SPECIFICATION
     PECIFICATION
INPUT
The_Child : Positive,
Of_The_Tree : Tree,
And_The_Tree : Tree
      OUTPUT
Of_The_Tree : Tree,
       And_The_Tree : Tree
EXCEPTIONS
```

```
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child Error
   OPERATOR Is_Equal
   SPECIFICATION
     INPUT
Left : Tree,
Right : Tree
     OUTPUT
Result : Boolean
EXCEPTIONS
DVerflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root, Child_Brror
   OPERATOR IS_Null SPECIFICATION
      INPUT
      The_Tree : Tree
OUTPUT
Result : Boolean
     EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root, Child_Error
  FND
  OPERATOR Item_Of SPECIFICATION
     INPUT
The_Tree : Tree
OUTPUT
Result : Item
EXCEPTIONS
LACEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
END
  OPERATOR Number_Of_Children_In
SPECIFICATION
INPUT
        The_Tree : Tree
     Result : Natural 
EXCEPTIONS
        Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END
  OPERATOR Child_Of
SPECIFICATION
INPUT
     The_Tree : Tree,
The_Child : Positive
OUTFUT
Result : Tree
      EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root, Child_Error
  END
IMPLEMENTATION ADA Tree_Arbitrary_Double_Unbounded_Managed
```

TREE ARBITRARY DOUBLE UNBOUNDED UNMANAGED

```
generic
     type Item is private;
Expected_Number_Of_Children : in Positive;
package Tree_Arbitrary_Double_Unbounded_Unmanaged is
     type Tree is private;
     Null_Tree : constant Tree;
                                                                     Tree;
                                (From The Tree
     procedure Copy
     procedure Clear (The_Tree procedure Clear (The_Iree procedure Construct (The_Item
                                                          : in out Tree);
: in out Tree);
: in Item;
: in out Tree;
    in Positive in out Tree; in out Tree);
                                 And_The_Tree
    modified by Tuan Nguyen
25 December 1995
adding procedures to replace functions
                                                            : in Tree;
: in Tree;
     procedure Is_Equal
                                               Right
                                               Result
(The_Tree
                                                               out Boolean);
                                                               in Tree;
out Boolean);
     procedure Is_Null
                                               Result
                                                             : in Tree);
                                               (The Tree
     procedure Item_Of
                                                               out Item);
```

```
procedure Number_Of_Children_In (The_Tree : in Tree;
                                                      (The_iree : in Tree;
Result : out Natural);
(The_Tree : in Tree;
The_Child : in Positive;
Result : out Tree);
      procedure Child_Of
-- end of modification
                                                     (Left
Right
                                                                     : in Tree;
: in Tree)
      function Is_Equal
                                                                                             return
Boolean;
function Is_Null
                                                                                             return
                                                     (The_Tree : in Tree)
Boolean:
      function Item_Of
                                                     (The_Tree : in Tree)
Item;
function Number_Of_Children_In (The_Tree : in Tree)
                                                                                             return
      function Child_Of
                                                    (The_Tree : in Tree;
The_Child : in Positive) return
Tree; function Parent_Of
                                                     (The_Tree : in Tree)
Tree;
      Overflow : exception;
Tree_Is_Null : exception;
Tree_Is_Not_Null : exception;
      Not_At_Root
Child_Error
                               : exception;
: exception;
type Node;
  type Tree is access Node;
  Null_Tree : constant Tree := null;
end Tree_Arbitrary_Double_Unbounded_Unmanaged;
```

TREE ARBITRARY DOUBLE UNBOUNDED UNMANAGED

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
                        "Restricted Rights Legend"
-- "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
---
with Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator;
package body Tree_Arbitrary_Double_Unbounded_Unmanaged is
       function Hash_Of (The_Child : in Positive) return Positive;
       package Children is new
    Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
    (Domain => Positive,
                 Hash Of
                                                  => Hash_Of);
        type Node is
               record
                      Previous : Tree;
The_Item : Item;
The_Children : Children.Map;
        function Hash_Of (The_Child : in Positive) return Positive is
       begin
return The_Child;
end Hash_Of;
       procedure Copy (From The Tree : in Tree;
To_The_Tree : in out Tree) is
procedure Copy_Child (The_Domain : in Positive;
The_Range : in Tree;
Continue : out Boolean) is
                      Temporary_Node : Tree;
               begin
              if From The Tree = null then
    To The Tree := null;
               else
              To_The_Tree := new Node;
To_The_Tree.The_Item := From_The_Tree.The_Item;
Copy_Children(From_The_Tree.The_Children);
end if;
        exception
when Storage_Error | Children.Overflow =>
raise Overflow;
        end Copy:
        procedure Clear (The_Tree : in out Tree) is
        begin
               The_Tree := null;
        end Clear;
       procedure Construct (The_Item : in And_The_Tree : in Number_Of_Children : in On_The_Child : in Temporary_Node : Tree;
                                                                                  : in Item;
: in out Tree;
                                                                                                   Natural:
                                                                                                   Natural) is
       begin
               if Number_Of_Children = 0 then
                       if And_The_Tree = null then
And_The_Tree := new Node;
And_The_Tree.The_Item := The_Item;
              return;
else
raise Tree_Is_Not_Null;
end if;
elsif On_The_Child > Number_Of_Children then
raise Child_Error;
elsif And_The_Tree = null then
And_The_Tree := new Node;
And_The_Tree.The_Item := The_Item;
for Index in 1 .. Number_Of_Children loop
Children.Bind(The_Domain => Index,
And_The_Range => null,
In_The_Map =>
                            return;
And_The_Tree.The_Children);
end loop;
elsif And_The_Tree.Previous = null then
```

```
Temporary_Node := new Node;
Temporary_Node.The_Item := The_Item;
for Index in 1 .. Number_Of_Children loop
    if Index = On_The_Child then
        Children_Bind
                               liter.Bind
(The_Domain => Index,
And_The_Range => And_The_Tree,
In_The_Map => Temporary_Node.The_Children);
                            Children Bind
                               Index. Sand
(The_Namin => Index,
And_The_Range => null,
In_The_Map => Temporary_Node.The_Children);
                       end if:
                 end 1:;
end loop;
And_The_Tree.Previous := Temporary_Node;
And_The_Tree := Temporary_Node;
           else
           raise Not_At_Root;
end if;
      exception
when Storage_Error | Children.Overflow =>
                raise Overflow;
     Of_The_Tree.The_Item := To_The_Item;
     exception
when Constraint_Error =>
raise Tree_Is_Null;
end Set_Item;
                                    (The_Child : in Positive
Of_The_Tree : in out Tree;
And_The_Tree : in out Tree) is
      procedure Swap_Child (The_Child
                                                                   Positive:
           Temporary_Node : Tree;
     begin
if And_The_Tree = null then
Temporary_Node := Children.Range_Of
(The_Domain => T
                                            (The_Domain => The_Child,
In_The_Map =>
Of_The_Tree.The_Children);
                 Children.Unbind(The_Child, Of_The_Tree.The_Children);
           else
           raise Not_At_Root; end if;
     end II;
exception
when Constraint_Error =>
raise Tree_Is_Null;
when Children.Domain_Is_Not_Bound =>
raise Child_Error;
end Swap_Child;
    modified by Tuan Nguyen
25 December 1995
adding procedures to replace functions
                                                   (Left
      procedure Is_Equal
                                                                  : in Tree:
                                                    Right
Result
                                                                  : in Tree;
: out Boolean) is
      begin
            Result := Is_Equal(Left, Right);
      end Is_Equal;
                                                   (The_Tree : in Tree;
Result : out Boolean) is
     procedure Is Null
           Result := Is_Null(The_Tree);
      end Is_Null;
                                                   (The_Tree : in Tree);
Result : out Item) is
      procedure Item_Of
      begin
      Result := Item_Of(The_Tree);
end Item_Of;
```

```
procedure Number_Of_Children_In (The_Tree : in Tree;
Result : out Natural) is
     Result := Number_Of_Children_In(The_Tree);
end Number_Of_Children_In;
                                                   (The_Tree : in Tree;
The_Child : in Positive;
Result : out Tree) is
     procedure Child_Of
     Result := Child_Of(The_Tree,The_Child);
end Child_Of;
-- end of modification
     in
if not Is_Equal(The_Range,
Children.Range_Of(The_Domain,
Right.The_Children))
then
                      Trees_Are_Equal := False;
Continue := False;
                 else
else
Continue := True;
end if;
end Check_Child_Equality;
procedure Check_Equality is new
Children.Iterate(Check_Child_Equality);
     begin
   if Left.The_Item /= Right.The_Item then
                return False;
           else
if Children.Extent_Of(Left.The_Children) /=
Children.Extent_Of(Right.The_Children) then
          else
Check_Equality(Left.The_Children);
return Trees_Are_Equal;
end if;
end if;
eption
      exception
```

```
when Constraint_Error =>
   return (Left = Null_Tree) and (Right = Null_Tree);
    function Is_Null (The_Tree : in Tree) return Boolean is
    begin
return (The_Tree = null);
end Is_Null;
    function Item_Of (The_Tree : in Tree) return Item is
    begin
return The_Tree.The_Item;
    exception
    when Constraint_Error =>
        raise Tree_Is_Null;
end Item_Of;
    function Number_Of_Children_In (The_Tree : in Tree) return Natural
    begin
         return Children.Extent_Of(The_Tree.The_Children);
    exception
when Constraint_Error =>
raise Tree_Is_Null;
end Number_Of_Children_In;
    begin
        exception
        when Constraint_Error =>
    raise Tree_Is_Null;
when Children_Domain_Is_Not_Bound =>
    raise Child_Error;
end Child_Of;
    function Parent_Of (The_Tree : in Tree) return Tree is
    begin
        return The_Tree.Previous;
    exception
when Constraint_Error =>
raise Tree_Is_Null;
end Parent_Of;
end Tree_Arbitrary_Double_Unbounded_Unmanaged;
```

TREE ARBITRARY DOUBLE UNBOUNDED UNMANAGED

PSDL

```
TYPE Tree_Arbitrary_Double_Unbounded_Unmanaged SPECIFICATION
   GENERIC
  Item : PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
      INPUT
        From_The_Tree : Tree,
To_The_Tree : Tree
     OUTPUT
      OUTPUT
TO_The_Tree : Tree
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
   END
  OPERATOR Clear SPECIFICATION
     INPUT
The_Tree : Tree
     OUTPUT
The_Tree : Tree
EXCEPTIONS
         Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  OPERATOR Construct
   SPECIFICATION
INPUT
The_Item : Item,
         And_The_Tree : Tree,
Number_Of_Children : Natural,
On_The_Child : Natural
     OUTPUT
And_The_Tree : Tree
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
   END
  OPERATOR Set_Item SPECIFICATION
      INPUT
Of_The_Tree : Tree,
To_The_Item : Item
      OUTPUT
      Of_The_Tree : Tree
EXCEPTIONS
         Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  OPERATOR Swap_Child
SPECIFICATION
INPUT
        NPUT
The_Child : Positive,
Of_The_Tree : Tree,
And_The_Tree : Tree
      OUTPUT
Of_The_Tree : Tree,
And_The_Tree : Tree
EXCEPTIONS
```

```
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child Error
   OPERATOR Is_Equal
   SPECIFICATION
INPUT
Left: Tree,
Right: Tree
     OUTPUT
Result : Boolean
      EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root, Child_Error
   OPERATOR IS_Null SPECIFICATION
     INPUT
The_Tree : Tree
OUTPUT
        Result : Boolean
      EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root, Child_Error
   END
   OPERATOR Item_Of SPECIFICATION
    INPUT The_Tree : Tree
OUTPUT
Result : Item
EXCEPTIONS
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
END
  OPERATOR Number_Of_Children_In
SPECIFICATION
INPUT
The_Tree : Tree
OUTFUT
Result : Natural
EXCEPTIONS
        Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
   OPERATOR Child_Of
SPECIFICATION
INFUT
The_Tree : Tree,
The_Child : Positive
OUTPUT
Result : Tree
      EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root, Child_Error
   END
IMPLEMENTATION ADA Tree_Arbitrary_Double_Unbounded_Unmanaged
```

TREE ARBITRARY SINGLE UNBOUNDED MANAGED

```
generic
   type Item is private;
   Expected_Number_Of_Children : in Positive;
package Tree_Arbitrary_Single_Unbounded_Managed is
      type Tree is private;
      Null_Tree : constant Tree;
     : in Tree;
: in out Tree);
: in out Tree);
                                                                         in Item;
in out Tree;
in Natural;
                                                                                    Natural):
                                                                      : in out Tree;
: in Item);
: in Positive;
                                        To The Item
      procedure Swap_Child (The_Child Of_The_Tree And_The_Tree
                                                                      : in out Tree;
: in out Tree);
     modified by Tuan Nguyen
25 December 1995
adding procedures to replace functions
                                                        (Left
                                                                         : in Tree;
      procedure Is_Equal
                                                                         : in Tree;
: out Boolean);
: in Tree;
                                                         Right
Result
                                                        (The_Tree
Result
(The_Tree
      procedure Is_Null
                                                                         : out Boolean);
: in Tree);
      procedure Item_Of
```

```
Result
      procedure Number_Of_Children_In (The_Tree : in Tree;
                                                      Result : out Natural);
(The_Tree : in Tree;
The_Child : in Positive;
      procedure Child_Of
                                                                      : out Tree);
                                                        Result
-- end of modification
      function Is_Equal
                                                     (Left
                                                                     : in Tree;
: in Tree)
                                                                                            return
Boolean:
                                                     (The_Tree : in Tree)
                                                                                           return
      function Is_Null
Boolean;
function Item_Of
                                                     (The_Tree : in Tree)
                                                                                            return
Item; function Number_Of_Children_In (The_Tree : in Tree)
                                                                                           return
Natural;
function Child_Of
                                                     (The_Tree : in Tree;
The_Child : in Positive) return
     Overflow : exception;
Tree_Is_Null : exception;
Tree_Is_Not_Null : exception;
Child_Error : exception;
private
  type Node;
  type Tree is access Node;
  Null_Tree : constant Tree := null;
end Tree_Arbitrary_Single_Unbounded_Managed;
```

TREE ARBITRARY SINGLE UNBOUNDED MANAGED

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch -- All Rights Reserved
-- Serial Number 0100219
   "Restricted Rights Legend"
Use, duplication, or disclosure is subject to restrictions as set forth in subdivision (b) (3) (ii)
- restrictions as set forth in Suddivision (7)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
with Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator,
Storage_Manager_Sequential;
package body Tree_Arbitrary_Single_Unbounded_Managed is
     function Hash_Of (The_Child : in Positive) return Positive;
     package Children is new
Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
(Domain => Positive,
            type Node is
           record
                ord
The_Item : Item;
The_Children : Children.Map;
Wayt : Tree;
      function Hash_Of (The_Child : in Positive) return Positive is
     return The_Child;
end Hash_Of;
     procedure Free (The_Node : in out Node) is begin
           Children.Clear(The_Node.The_Children);
     end Free;
     The_Node.Next := To_Next;
end Set_Next;
      function Next_Of (The_Node : in Node) return Tree is
     return The_Node.Next;
end Next_Of;
     Free
                                                                    => Free
                                                    Set Pointer => Set_Next,
                                                   Pointer_Of
     procedure Copy (From The Tree : in Tree;
To_The Tree : in out Tree) is
procedure Copy_Child (The Domain : in Positive;
The_Range : in Tree;
Continue : out Boolean) is
                Temporary_Node : Tree;
          end Copy_Child;
procedure Copy_Children is new Children.Iterate(Copy_Child);
begin
   Clear(To_The_Tree);
if From_The_Tree /= null then
    To_The_Tree := Node_Manager.New_Item;
   To_The_Tree.The_Item := From_The_Tree.The_Item;
   Copy_Children(From_The_Tree.The_Children);
end if:
           end if:
      exception
when Storage_Error | Children.Overflow =>
raise Overflow;
      end Copy;
     Temporary_Node : Tree
                                               := The_Range;
           begin
Clear(Temporary_Node);
           Continue := True;
end Clear_Child;
procedure Clear_Children is new Children.Iterate(Clear_Child);
     begin
```

```
if The_Tree /= null then
   Clear_Children(The_Tree.The_Children);
   Node_Manager.Free(The_Tree);
         end if:
  end Clear:
 procedure Construct (The_Item
                                   And_The_Tree : in out Tree;
Number_Of_Children : in Natura
On_The_Child : in Natura
                                                                                 Natural:
        Temporary_Node : Tree;
 begin
        in
if Number_Of_Children = 0 then
if And_The_Tree = null then
And_The_Tree := Node_Manager.New_Item;
And_The_Tree.The_Item := The_Item;
               else
        raise Tree_Is_Not_Null;
end if;
elsif On_The_Child > Number_Of_Children then
raise Child_Error;
        else
              Temporary_Node := Node_Manager.New_Item;
Temporary_Node.The_Item := The_Item;
for Index in 1 .. Number_Of_Children loop
    if Index = On_The_Child then
        Children.Bind
                                Indexn.Bind
(The_Domain => Index,
And_The_Range => And_The_Tree,
In The_Map => Temporary_Node.The_Children);
                               (The_Domain
                     else
Children.Bind
                                                      => Index,
                                And_The_Range => null,
In_The_Map => Temporary_Node.The_Children);
               end if;
end loop;
              And The Tree := Temporary Node:
        end if;
  exception
when Storage_Error | Children.Overflow =>
raise Overflow;
end Construct;
 Of_The_Tree.The_Item := To_The_Item;
  exception
when Constraint_Error =>
raise Tree_Is_Null;
end Set_Item;
 procedure Swap_Child (The_Child : in Positive; Of_The_Tree : in out Tree; And_The_Tree : in out Tree) is
        Temporary_Node : Tree;
       and_The_Range => And_T
In_The_Map => Of_Th
And_The_Tree := Temporary_Node;
exception
when Constraint_Error =>
raise Tree_Is_Null;
when Children.Domain_Is_Not_Bound =>
raise Child_Error;
end Swap_Child;
modification
 modified by Tuan Nguyen
25 December 1995
 adding procedures to replace functions
                                                                        : in Tree;
: in Tree;
: out Boolean) is
 procedure Is_Equal
                                                      Left
                                                       Right
Result
  Result := Is_Equal(Left,Right);
end Is_Equal;
  procedure Is_Null
                                                      (The_Tree : in Tree;
                                                       Result
                                                                        : out Boolean) is
 begin
        Result := Is_Null(The_Tree);
  end Is_Null;
 procedure Item_Of
                                                     (The_Tree : in Tree);
Result : out Item) is
  Result := Item_Of(The_Tree);
end Item_Of;
  procedure Number_Of_Children_In (The_Tree : in Tree;
Result : out Natural) is
```

```
begin
    Result := Number_Of_Children_In(The_Tree);
end Number_Of_Children_In;
                                      (The_Tree : in Tree;
The_Child : in Positive;
Result : out Tree) is
    procedure Child_Of
    begin
        Result := Child_Of(The_Tree,The_Child);
    end Child Of:
   -- end of modification
        then
                Trees_Are_Equal := False;
Continue := False;
            else
begin
        if Left.The_Item /= Right.The_Item then
    return False;
        else
            if Children.Extent_Of(Left.The_Children) /=
    Children.Extent_Of(Right.The_Children) then
    return False;
            else
                Check_Equality(Left.The_Children);
return Trees_Are_Equal;
```

```
end if;
end if;
    exception
when Constraint_Error =>
return (Left = Null_Tree) and (Right = Null_Tree);
end Is_Equal;
    function Is_Null (The_Tree : in Tree) return Boolean is
    return (The_Tree = null);
end Is_Null;
    function Item_Of (The_Tree : in Tree) return Item is
    begin return The_Tree.The_Item;
    exception
when Constraint_Error =>
raise Tree_Is_Null;
end Item_Of;
    function Number_Of_Children_In (The_Tree : in Tree) return Natural
        return Children.Extent_Of(The_Tree.The_Children);
    exception
when Constraint_Error =>
    raise Tree_Is_Null;
end Number_Of_Children_In;
    begin
        exception
   when Constraint_Error =>
        raise Tree_Is_Null;
   when Children_Domain_Is_Not_Bound =>
        raise Child_Error;
end Child_Of;
end Tree_Arbitrary_Single_Unbounded_Managed;
```

TREE ARBITRARY SINGLE UNBOUNDED MANAGED

PSDL

```
TYPE Tree_Arbitrary_Single_Unbounded_Managed SPECIFICATION
   GENERIC
Item: PRIVATE_TYPE
OPERATOR COPY
SPECIFICATION
       PRCIFICATION
INPUT
From_The_Tree : Tree,
To_The_Tree : Tree
OUTFUT
To_The_Tree : Tree
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
ND
   OPERATOR Clear
SPECIFICATION
       INPUT
       The_Tree : Tree
           The Tree : Tree
       EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
    END
   OPERATOR Construct
SPECIFICATION
       INPUT
       INFUT
The_Item : Item,
And_The_Tree : Tree,
Number_Of_Children : Natural,
On_The_Child : Natural
OUTPUT
And_The_Tree : Tree
EXCEPTIONS
           Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
    END
   OPERATOR Set_Item
SPECIFICATION
INPUT
Of_The_Tree : Tree,
To_The_Item : Item
       OUTPUT
Of_The_Tree : Tree
EXCEPTIONS
           Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
    END
    OPERATOR Swap_Child
SPECIFICATION
       PECIFICATION
INPUT
The_Child : Positive,
Of_The_Tree : Tree,
And_The_Tree : Tree
       OUTPUT
Of_The_Tree : Tree,
And The_Tree : Tree
EXCEPTIONS
```

```
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  OPERATOR Is_Equal
SPECIFICATION
INPUT
Left : Tree,
     Right : Tree
OUTPUT
Result : Boolean
     EXCEPTIONS
        Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  OPERATOR Is_Null
SPECIFICATION
INFUT
The_Tree: Tree
OUTFUT
Result: Boolean
EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  OPERATOR Item_Of
SPECIFICATION
INPUT
The_Tree : Tree
OUTPUT
Result : Item
     EXCEPTIONS
        Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  OPERATOR Number_Of_Children_In
SPECIFICATION
INPUT
The_Tree : Tree
     OUTPUT
     Result : Natural EXCEPTIONS
        Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  OPERATOR Child_Of SPECIFICATION
     The_Tree : Tree,
The_Child : Positive
OUTPUT
     Result : Tree
EXCEPTIONS
        Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  EMD
IMPLEMENTATION ADA Tree_Arbitrary_Single_Unbounded_Managed
```

TREE ARBITRARY SINGLE UNBOUNDED UNMANAGED

```
generic
generic
type Item is private;
Expected_Number_Of_Children : in Positive;
package Tree_Arbitrary_Single_Unbounded_Unmanaged is
                         type Tree is private;
                       Null Tree : constant Tree:
                     procedure Copy (From_The_Tree : in out Tree;

procedure Clear (The_Tree : in out Tree);

(The_Tree : in out Tree);

(The_Item : in Item;

And_The_Tree : in out Tree;

And_The_Tree : in out Tree;

Number_Of_Children : in out Tree;

On_The_Child : in Natural;

procedure Set_Item (Of_The_Tree : in out Tree;

The_Item : in Item;

Ite
                                                                                                                                                                                                                                                                                                                                                      Natural);
                                                                                                                                                                                                                                                                                             : in out Tree;
: in Item);
                                                                                                                                                                   To The Item
                      procedure Swap_Child (The_Child Of_The_Tree And_The_Tree
                                                                                                                                                                                                                                                                                                                                                      Positive;
                                                                                                                                                                                                                                                                                             : in
                                                                                                                                                                                                                                                                                            : in out Tree;
: in out Tree);
                     modified by Tuan Nguyen
25 December 1995
adding procedures to replace functions
                                                                                                                                                                                                                                     (Left
                                                                                                                                                                                                                                                                                                           : in Tree;
                         procedure Is_Equal
                                                                                                                                                                                                                                     Right
Result
(The_Tree
Result
                                                                                                                                                                                                                                                                                                            : in Tree;
                                                                                                                                                                                                                                                                                                         : in Tree;
: out Boolean);
: in Tree;
: out Boolean);
: in Tree);
                         procedure Is_Null
                         procedure Item_Of
```

```
: out Item);
: in Tree;
                                                        Result
                                                                        : in Tree;
: out Natural);
: in Tree;
      procedure Number_Of_Children_In (The_Tree Result
                                                        (The_Tree
      procedure Child_Of
                                                       The_Child : in Positive;
Result : out Tree);
-- end of modification
                                                                      : in Tree;
: in Tree)
      function Is_Equal
                                                       Right
Boolean;
      function Is_Null
                                                      (The_Tree : in Tree)
                                                                                             return
Boolean;
function Item_Of
                                                      (The_Tree : in Tree)
                                                                                             return
       function Number_Of_Children_In (The_Tree : in Tree)
Natural;
function Child_Of
                                                      (The_Tree : in Tree;
The_Child : in Positive) return
      Overflow : exception;
Tree_Is_Null : exception;
Tree_Is_Not_Null : exception;
Child_Error : exception;
private
private
  type Node;
  type Tree is access Node;
  Null_Tree : constant Tree := null;
end Tree_Arbitrary_Single_Unbounded_Unmanaged;
```

TREE ARRITRARY SINGLE UNROUNDED UNMANAGED

```
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
-- Serial Number 0100219
"Restricted Rights Legend"
- Use, duplication, or disclosure is subject to
- restrictions as set forth in subdivision (b) (3) (ii)
- of the rights in Technical Data and Computer
- Software Clause of FAR 52.227-7013. Manufacturer:
- Wizard software, 2171 S. Parfet Court, Lakewood,
- Colorado 80227 (1-303-987-1874)
with Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator; package body Tree_Arbitrary_Single_Unbounded_Unmanaged is
        function Hash_Of (The_Child : in Positive) return Positive;
       package Children is new
           Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
(Domain => Positive,
                 type Node is
               record
The_Item : Item;
The_Children : Children.Map;
               end record;
        function Hash_Of (The_Child : in Positive) return Positive is
       begin
               return The Child:
       end Hash_Of;
       procedure Copy (From The_Tree : in Tree;
To_The_Tree : in out Tree) is
procedure Copy_Child (The_Domain : in Positive;
The_Range : in Tree;
Continue : out Boolean) is
                       Temporary_Node : Tree;
              Temporary_Node : Itee,
begin
    Copy(The_Range, To_The_Tree => Temporary_Node);
    Children.Bind(The_Domain, Temporary_Node,
    In_The_Map => To_The_Tree.The_Children);
    Continue := True;
end Copy_Child;
procedure Copy_Children is new Children.Iterate(Copy_Child);
               if From_The_Tree = null then
    To_The_Tree := null;
                       To_The_Tree := new Node;
To_The_Tree.The_Item := From_The_Tree.The_Item;
Copy_Children(From_The_Tree.The_Children);
               end if:
       exception
when Storage_Error | Children.Overflow =>
raise Overflow;
        end Copy:
       procedure Clear (The_Tree : in out Tree) is
begin
               The Tree := null:
        end Clear:
       procedure Construct (The_Item : in And_The_Tree : in Number_Of_Children : in On_The_Child : in Temporary_Node : Tree;
                                                                                      : in Item;
: in out Tree;
                                                                                                         Natural;
       begin
               if Number_Of_Children = 0 then
   if And_The_Tree := null then
   And_The_Tree := new Node;
   And_The_Tree.The_Item := The_Item;
                              return;
                       raise Tree_Is_Not_Null; end if;
               end 1r;
elsif On_The_Child > Number_Of_Children then
  raise Child_Error;
                else
                       Temporary_Node := new Node;
Temporary_Node.The_Item := The_Item;
for Index in 1 . . Number_Of_Children loop
if Index = On_The_Child then
Children.Bind

'mth Pomain => Index.
                                            Ildren.Bind (The_Domain => Index,
And_The_Range => And_The_Tree,
In_The_Map => Temporary_Node.The_Children);
                                else
                                        .
Children.Bind
                                            The_Nomain => Index,
And_The_Range => null,
In_The_Map => Temporary_Node.The_Children);
```

```
end if:
              end 11;
end loop;
And_The_Tree := Temporary_Node;
         end if:
     exception
when Storage_Error | Children.Overflow =>
raise Overflow;
    end Construct:
    procedure Set_Item (Of_The_Tree : in out Tree;
To_The_Item : in Item) is
         Of_The_Tree.The_Item := To_The_Item;
    exception when Constraint_Error =>
    raise Tree_Is_Null;
end Set_Item;
    procedure Swap_Child (The_Child : in Positive Of_The_Tree : in out Tree; And_The_Tree : in out Tree) is
                                                        Positive:
         Temporary_Node : Tree;
        exception when Constraint_Error
    raise Tree_Is_Null;
when Children.Domain_Is_Not_Bound =>
raise Child_Error;
end Swap_Child;
    modified by Tuan Nguyen
25 December 1995
    adding procedures to replace functions
                                           (Left
                                                       : in Tree;
: in Tree;
    procedure Is Equal
                                            Right
                                                       : out Boolean) is
                                            Result
    begin
         Result := Is_Equal(Left,Right);
    end Is_Equal;
                                           (The_Tree : in Tree;
Result : out Boolean) is
    procedure Is Null
         Result := Is_Null(The_Tree);
    end Is_Null;
                                           (The_Tree : in Tree);
Result : out Item) is
    procedure Item Of
         Result := Item_Of(The_Tree);
    end Item_Of;
    begin
         Result := Number Of Children In(The Tree);
     end Number_Of_Children_In;
                                           (The_Tree : in Tree;
The_Child : in Positive;
    procedure Child_Of
                                            Result
                                                       : out Tree) is
    begin
         Result := Child_Of(The_Tree,The_Child);
    end Child Of;
    in
if not Is_Equal{The_Range,
Children.Range_Of{The_Domain,
Right.The_C
                                                      Right . The Children ) )
then
                  Trees_Are_Equal := False;
Continue := False;
continue := False;
else
continue := True;
end if;
end Check_Child_Equality;
procedure Check_Equality is new
Children.Iterate(Check_Child_Equality);
    begin
   if Left.The_Item /= Right.The_Item then
```

```
else
    if Children.Extent_Of(Left.The_Children) /=
        Children.Extent_Of(Right.The_Children) /=
        Children.Extent_Of(Right.The_Children) then
        return False;

else
        Check_Equality(Left.The_Children);
        return Trees_Are_Equal;
    end if;
end if;
exception
    when Constraint_Error =>
        return (Left = Null_Tree) and (Right = Null_Tree);
end Is_Equal;
function Is_Null (The_Tree : in Tree) return Boolean is begin
    return (The_Tree = null);
end Is_Null;
function Item_Of (The_Tree : in Tree) return Item is begin
    return The_Tree.The_Item;
exception
    when Constraint_Error =>
        raise Tree_Is_Null;
```

TREE ARBITRARY SINGLE UNBOUNDED UNMANAGED

PSDL

```
TYPE Tree_Arbitrary_Single_Unbounded_Unmanaged SPECIFICATION
   GENERIC
   GENERIC
Item: PRIVATE_TYPE
OPERATOR Copy
SPECIFICATION
       From The Tree : Tree,
To The Tree : Tree
OUTPUT
       To_The_Tree : Tree EXCEPTIONS
          Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
   END
   OPERATOR Clear
SPECIFICATION
       INPUT
The_Tree : Tree
       OUTPUT
           The_Tree : Tree
          Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
   OPERATOR Construct
SPECIFICATION
      PECIFICATION
INPUT
The_Item : Item,
And_The_Tree : Tree,
Number_Of_Children : Natural,
On_The_Child : Natural
OUTFUT
And_The_Tree : Tree
EXCEPTIONS
           Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
    END
   OPERATOR Set_Item
SPECIFICATION
INFUT
Of_The_Tree : Tree,
To_The_Item : Item
OUTPUT
Of_The_Tree : Tree
PYCPETIONS
       EXCEPTIONS
           Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
   OPERATOR Swap_Child
SPECIFICATION
INPUT
The_Child: Positive,
Of The_Tree: Tree,
And_The_Tree: Tree
OUTPUT
Of_The_Tree: Tree,
And_The_Tree: Tree
EXCEPTIONS
```

```
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END
  OPERATOR Is_Equal SPECIFICATION
    INPUT
Left : Tree,
Right : Tree
    OUTPUT
    Result : Boolean
EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  EMD
  OPERATOR Is_Null SPECIFICATION
    INPUT
The_Tree : Tree
OUTPUT
    Result : Boolean
EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END
  OPERATOR Item_Of SPECIFICATION
    INPUT
The_Tree : Tree
    OUTPUT
      Result : Item
    EXCEPTIONS
Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  OPERATOR Number_Of_Children_In SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
Result : Natural
EXCEPTIONS
       Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  OPERATOR Child_Of
  SPECIFICATION
INPUT
The_Tree : Tree,
    The_Child : Positive
OUTPUT
    Result : Tree
       Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END
IMPLEMENTATION ADA Tree_Arbitrary_Single_Unbounded_Unmanaged END
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center 8725 John J. Kingman Road, Suite 0944 Fort Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Road Monterey, CA 93943	2
3.	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
4.	Dr. Ted Lewis, Chairman, Code CS/L Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
5.	Chief of Naval Research 800 N. Quincy Street Arlington, VA 22217	1
6.	Dr. Luqi, Code CS/Lq Naval Postgraduate School Computer Science Department Monterey, CA 93943-5100	20
7.	LT Tuan A. Nguyen 3673 Henrico Street Norfolk, VA 23513	3
8.	Ada Joint Program Office OUSDRE (R&AT) The Pentagon Washington, DC 20301	1
9.	Atlantic Intelligence Command 7941 Blandy Road Suite 100 Norfolk, VA 23551	1